US Army Corps
of Engineers
Construction Engineering
Research Laboratories

AD-A273 355

USACERL Technical Report FF-93/11
September 1993

(2)

Definition and Implementation of the Integrated Modular Persistent Object RepresentationTranslator (IMPORT)

by
Charles E. Herring
Joseph Teo
Vijay Karamcheti
R. Alan Whitehurst
Biju L. Kalathil
Heien-Kun Chiang
John Pietrzak

The Integrated Modular Persistent Object Representation Translator (IMPORT) is a programming language developed as part the Integrated Systems Language Environment (ISLE), and intended to provide software engineering support for modeling and simulation of complex ecological systems such as the modern battlefield. IMPORT integrates a number of software technologies: object-oriented imperative programming, knowledge-based declarative programming, process-based simulation, and persistent object storage.

IMPORT manages programs as data, like a CAD system manipulates design artifacts. It is implemented as an object-oriented framework that models the language and supports persistent object storage of an intermediate representation in the form of abstract syntax trees, symbol tables, and associated structures. The parser translates input into this intermediate representation, and subsequent operations on these artifacts occur through the classes of the framework. Facilities for editing, browsing, compiling, version control, interpreting/debugging, optimizing, code generating, and profiling are provided based on the framework. This report defines the language, describes its implementation, and outlines research directions within the context of the project.

The contents of this report are not to be used for advertising, publication, or promotional purposes. Citation of trade names does not constitute an official endorsement or approval of the use of such commercial products. The findings of this report are not to be construed as an official Department of the Army position, unless so designated by other authorized documents.

DESTROY THIS REPORT WHEN IT IS NO LONGER NEEDED

DO NOT RETURN IT TO THE ORIGINATOR

USER EVALUATION OF REPORT

REFERENCE: USACERL Technical Report (TR) FF-93/11, *Definition and Implementation of the Integrated Modular Persistent Object Representation Translator (IMPORT)*

Please take a few minutes to answer the questions below, tear out this sheet, and return it to USACERL. As user of this report, your customer comments will provide USACERL with information essential for improving future reports.

1. Does this report satisfy a need? (Comment on purpose, related project, or other area of interest for which report will be used.)

_____

_____

_____

2. How, specifically, is the report being used? (Information source, design data or procedure, management procedure, source of ideas, etc.)

_____

_____

3. Has the information in this report led to any quantitative savings as far as manhours/contract dollars saved, operating costs avoided, efficiencies achieved, etc.? If so, please elaborate.

_____

_____

4. What is your evaluation of this report in the following areas?

   a. Presentation:_____

   b. Completeness:_____

   c. Easy to Understand:_____

   d. Easy to Implement:_____

   e. Adequate Reference Material:_____

   f. Relates to Area of Interest:_____

   g. Did the report meet your expectations?_____

   h. Does the report raise unanswered questions?_____

i. General Comments. (Indicate what you think should be changed to make this report and future reports of this type more responsive to your needs, more usable, improve readability, etc.)

_____

_____

_____

_____

_____

5. If you would like to be contacted by the personnel who prepared this report to raise specific questions or discuss the topic, please fill in the following information.

Name: _____

Telephone Number: _____

Organization Address: _____

_____

_____

6. Please mail the completed form to:

Department of the Army
CONSTRUCTION ENGINEERING RESEARCH LABORATORIES
ATTN: CECER-IMT
P.O. Box 9005
Champaign, IL 61826-9005

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. AGENCY USE ONLY (Leave Blank)	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	September 1993	Final

4. TITLE AND SUBTITLE
Definition and Implementation of the Integrated Modular Persistent Object Representation Translator (IMPORT)

5. FUNDING NUMBERS
4A162784
AT41
FZ-AV3

6. AUTHOR(S)
Charles E. Herring, Joseph Teo, Vijay Karamcheti, R. Alan Whitehurst, Biju L. Kalathil, Heien-Kun Chiang, and John Pietrzak

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
U.S. Army Construction Engineering Research Laboratories (USACERL)
P.O. Box 9005
Champaign, IL 61826-9005

8. PERFORMING ORGANIZATION REPORT NUMBER
TR-FF-93/11

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)
Office of the Chief of Engineers
ATTN: DAEN-ZCM
Room 1E682 The Pentagon
Washington, DC 20310-2600

10. SPONSORING/MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES
Copies are available from the National Technical Information Service, 5285 Port Royal Road, Springfield, VA 22161.

12a. DISTRIBUTION/AVAILABILITY STATEMENT
Approved for public release; distribution is unlimited.

12b. DISTRIBUTION CODE

13. ABSTRACT (Maximum 200 words)

The Integrated Modular Persistent Object Representation Translator (IMPORT) is a programming language developed as part the Integrated Systems Language Environment (ISLE), and intended to provide software engineering support for modeling and simulation of complex ecological systems such as the modern battlefield. IMPORT integrates a number of software technologies: object-oriented imperative programming, knowledge-based declarative programming, process-based simulation, and persistent object storage.

IMPORT manages programs as data, like a CAD system manipulates design artifacts. It is implemented as an object-oriented framework that models the language and supports persistent object storage of an intermediate representation in the form of abstract syntax trees, symbol tables, and associated structures. The parser translates input into this intermediate representation, and subsequent operations on these artifacts occur through the classes of the framework. Facilities for editing, browsing, compiling, version control, interpreting/debugging, optimizing, code generating, and profiling are provided based on the framework. This report defines the language, describes its implementation, and outlines research directions within the context of the project.

14. SUBJECT TERMS
Integrated Modular Persistent Object Representation Translator (IMPORT)
modeling simulation object-oriented programming
integrated systems language environment ecological systems

15. NUMBER OF PAGES
146

16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
Unclassified	Unclassified	Unclassified	SAR

FOREWORD

This study was conducted for the Office of the Chief of Engineers (OCE), Military Engineering and Topography Division, under Project 4A162784AT41, "Military Facilities Engineering Technology"; Work Unit FZ-AV3, "Integrated Modular Persistent Object Representation Translator (IMPORT)." The technical monitor was David Loental, U.S Army Engineer School (USAES-DCD).

This research was done by the Facility Management Division (FF), Infrastructure Laboratory (FL), U.S. Army Construction Engineering Research Laboratories (USACERL). Gratitude is expressed to Walter Hollis, Deputy Undersecretary of the Army for Operations Research, and to the SimTech Committee and the staff at the U.S. Army Model and Simulation Office, especially COL Gilbert Brauch, Director. Dr. Janet H. Spoonamore is Acting Chief, CECER-FF, and Dr. Michael J. O'Connor is Chief, CECER-FL. The USACERL technical editor was William J. Wolfe, Information Management Office.

LTC David J. Rehbein is Commander of USACERL and Dr. L.R. Shaffer is Director.

CONTENTS

FIGURES

TABLES

DEFINITION AND IMPLEMENTATION OF THE INTEGRATED MODULAR OBJECT REPRESENTATION TRANSLATOR (IMPORT)

1 INTRODUCTION

Background

Scientific and engineering disciplines, from computer science to operations research, have long used computer simulations to develop models that help analysts understand, predict, and control complex systems. Over time, a large body of knowledge, methodologies, and software tools have been developed for domain specifics and to meet general purpose needs. These approaches typically rely on the application of a single software technology, such as imperative programming, to derive a software system the sole function of which is simulation. This approach has been effective for simulation of isolated systems.

Over the past two decades, many efforts have been made to develop large-scale simulations spanning many levels of system resolution, e.g., the combat models developed by operations researchers in support of defense requirements (Davis and Huber 1992). These efforts to develop simulations of large parts of reality have used most software technologies and techniques (Herring, Wallace, and Whitehurst 1991). For example, declarative programming is typically used to model human decisionmaking, while, for efficiency, static portions of models are written in imperative languages. However, most expert system shells are designed as standalone packages, providing only external interfaces to other programming systems. Other similarly designed software technologies cannot address modeling and simulation in a consistent and integrated manner. Recent Advanced Research Projects Agency (ARPA) and Department of Defense (DOD) initiatives have expressed goals far beyond the use of analytical simulations as single-user tools (DA 1992, Director of Defense Research and Engineering 1992a, 1992b). These initiatives have piloted research on Distributed Interactive Simulation (DIS) to use virtual reality for training, hardware prototyping and evaluation, as well as for analytical purposes.

In recognition of the trend toward object-oriented programming and its appropriateness for modeling and simulation, the U.S. Army sponsored the development of an object-oriented programming language for simulation. In 1988, this project began to investigate the integration of modern software and hardware architectures to support large-scale simulation. This resulted in the U.S. Army ModSim, Version 1.0, the modular simulation language (Herring 1990). ModSim is a general-purpose fully object-oriented programming language based on Modula-2, which provides strong typing and modularity for programming in the large (Wirth 1984). ModSim also provides an object data type and integrates process-based discrete-event simulation with objects. Methods of objects are asynchronous threads of execution and the language provides primitives for passing simulation time and for synchronization.

As a research test bed, a combat model was developed using ModSim to experiment with the application of software technologies and techniques. The Model-View-Controller framework of SMALLTALK (Krasner 1988) was used as a guide to designing both the global and local architecture of the model. From a study of existing models and knowledge of emerging software capabilities, two requirements became evident: (1) the need for declarative programming to model complex decision-making, and (2) the need for consistent object storage for model management. Work on these requirements resulted in the ModLog (modular logic) declarative programming language (Whitehurst 1991, 1992), and Persistent ModSim (Herring 1991, 1993).

Based on experience with these two languages, and with other prototypes during applications development, a new language was designed and implemented to provide full integration of the software technologies identified to support general modeling and simulation, the IMPORT/DOME language system. IMPORT/DOME is one of a number of tools that comprise the Integrated Simulation Language Environment (ISLE). The tools in the ISLE environment interact with the persistent object repository, which stores information about the class hierarchy, the programs under development, and the results of program execution—all at the object-level of granularity (as opposed to the file, or function-level granularities that exist in most systems). All the tools in ISLE interact with the object repository and operate from the intermediate form of IMPORT/DOME programs.

Objective

The objective of this work was to develop an integrated application of an object-oriented, imperative and declarative programming language that combined process-based discrete-event simulation and persistent object storage to address large-scale, complex systems modeling and simulation.

Approach

IMPORT is a direct descendent of ModSim, and Persistent ModSim is a declarative language extension to ModSim, based on PROLOG, which is a language developed for theorem proving and artificial intelligence applications. Persistent ModSim differs from PROLOG in that it was intended to provided intelligent decisionmaking support to objects while embedded in a simulation; it is designed for solving problems that involve objects and the relationships between objects. The Persistent ModSim extension to ModSim consists of a language for specifying declarative rule bases, an object-oriented interpreter for the language that can be inherited by objects, and a facility for "binding" knowledge-base objects to the imperative objects in the simulation.

Persistent ModSim is a version of ModSim integrated with a commercial object-oriented database. The approach taken to provide ModSim with persistent object storage capabilities was quite straightforward. Two changes were required in the syntax: (1) the NEWOBJ primitive was extended to include an optional second parameter, which is an object reference to a database, configuration, or segment. This permits specification of object instance allocation in the persistent object store; (2) a TRANSACTION statement was introduced to provide for short term database transaction management. This was necessary to provide multi-user access to the same database files. Next a set of classes were developed to provide access to the underlying database functionality, including: database, database root, collection, configuration, workspace, and segment. As the object-oriented database supports a C language library interface and is implemented as a translator to C, minor changes in the C code generation provide for interfacing to the database.

IMPORT differs significantly from ModSim in that it was specifically designed for:

1. A higher degree of integration and consistency with the syntax and semantics of the declarative complementary language.

2. Enhancement of the language with new constructs and removal of those deemed unnecessary.

3. An implementation based on the persistent object repository, providing for storage of the intermediate object representation and versioning.

IMPORT provides for intermodule-type definition visibility, and is integrated in several ways: (1) it provides for the integrated application of object-oriented imperative programming and process-based simulation as in the model; (2) it is integrated with DOME (as described below); (3) it is integrated in its unique implementation, which provides for the development of an integrated set of software engineering support tools. IMPORT has four module types: it separates interface specification and implementation, for the INTERFACE module type for interfacing to other languages, and the KNOWLEDGE module for integration with DOME.

Mode of Technology Transfer

U.S. Army ModSim, Version 1.0 has been distributed to Government agencies and their contractors along with a suite of example programs that aid in understanding the process-based simulation model of ModSim and general object-oriented programming, as well a graphical class-hierarchy browser and an editing/compilation management environment. It is anticipated that the completed program may be distributed cooperatively through several particpating agencies: the Army Material Systems Analysis Agency, the Defense Logistics Agency, and through contractors on the Advanced Research Project Agency's next generation Distributed Interactive Simulation (DIS) War Breaker (Booze-Allen-Hamilton, Science Applications International Corporation, and The Applications Science Corporation).

2 LANGUAGE DEFINITION

This definition of the IMPORT language is provided for those who wish to study the details of the language, primarily those interested in evaluating it for use or as a reference for programming. A more compact syntactic description, known as a context free grammar, is given in Appendix A.

Introduction

IMPORT retains the DEFINITION and IMPLEMENTATION modules of its predecessors, but has the two new module types: the KNOWLEDGE module and the INTERFACE module. The KNOWLEDGE module is an approach to providing a more consistent integration between IMPORT and DOME. It contains the declarative programming language statements of DOME as QUERY methods and relates them to IMPORT objects defined in DEFINITION modules and to the method bodies that are located in IMPLEMENTATION modules. The LIST type permits passing parameters between the IMPORT and DOME languages. Enumeration type variables are also parameters to DOME QUERY methods.

ModSim introduced the ASK and TELL methods. TELL methods are asynchronous and have time-passing statements (such as WAIT DURATION), whereas ASK methods are synchronous and cannot have simulation time passing statements. IMPORT removes the restriction on ASK methods. If they have time-passing statements, they will be synchronous, that is, the program will wait for their completion. This permits passing parameters by reference to ASK methods. QUERY methods appear in object definitions just as the imperative methods. IMPORT extends object methods over ModSim in a number of ways. There are OPERATOR methods providing for operator overloading. (All methods may have overloaded signatures with the exception of DESTRUCTOR methods.) Additionally, overloaded CONSTRUCTOR methods may be specified through extended syntax of the NEW primitive.

The following have been removed from IMPORT: records, procedures, pointers, and subrange types. The removal of procedures caused the MAIN module construct for specifying the starting point of an application to become inconsistent, so it was replaced by the KEY statement, which is used to specify applications. The KEY specifies an object to be created, and a method of the object that 1will be the entry point of the application. KEYs are used by the interpreter and code generator to determine the modules and objects involved in an application. A KEY statement has the form:

KEY *key_name*
FROM *module_name* CREATE *class_name*
INVOKE *method_name (arguments).*

Persistence in IMPORT is achieved as in Persistent ModSim through overloading the NEW function and providing a set of classes that model a generic object-oriented database capability.

There are other minor, but convenient changes. For example, the language is case insensitive to keywords, and identifiers cannot be keywords. The familiar C shorthand assignment operators (+=, -=, etc.) are supported.

Definition of Terms

The following description uses conventional terminology. A *variable* is a place holder or *location* for the result of a computation. The result of a computation must have a defined domain, called a *type*. Variables must also have type, and can only hold values within that domain.

An *identifier* is a symbol declared as a name for a variable, a type, a constant, a class, etc. The scope of a declaration is the region over which a declaration has effect. IMPORT is a lexically scoped language. An *expression* specifies a computation which produces a value. The term *repository* refers to the object-oriented database and associated schemas used to store IMPORT modules.

Modules

The highest level of abstraction characterizing IMPORT is the *module*. An IMPORT application is a collection of modules and a *key*. There are four types of modules: DEFINITION modules, IMPLEMENTATION modules, KNOWLEDGE modules, and INTERFACE modules.

Each IMPLEMENTATION module, KNOWLEDGE module, and INTERFACE module must have a corresponding DEFINITION module of the same name. We may refer to these three modules collectively as *implementation* modules. Each DEFINITION module must have either an IMPLEMENTATION module, or a KNOWLEDGE module, or an INTERFACE module or any combination of the three. The relation of modules in IMPORT is shown in Figure 1.

The declarations and IMPORTs made in a DEFINITION module are automatically included into the corresponding IMPLEMENTATION, KNOWLEDGE, and INTERFACE modules.

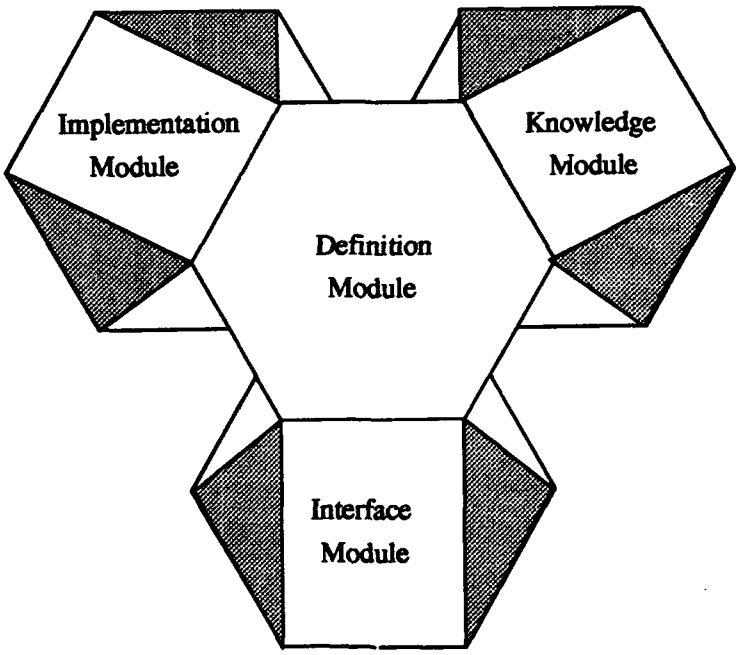

Figure 1. Relationships of IMPORT Modules.

DEFINITION Modules

A DEFINITION module contains *type* and *class* definitions. It also contains any global *constants* that may be used in implementation modules. These modules define the interface to be used with the implementation modules.

Data sharing between modules is achieved via an IMPORT statement, which is described in **Statements**, p 18.

A DEFINITION module has the following form:

DEFINITION MODULE *module_name*;
IMPORT *statements*
    type, constant and class definition list
END MODULE.

IMPLEMENTATION Modules

IMPLEMENTATION modules contain the implementation and body of the classes defined in the DEFINITION module. These classes contain imperative (procedural) code.

An IMPLEMENTATION module has the following form:

IMPLEMENTATION MODULE *module_name;*
IMPORT *statements*
    class$_1$ imperative and declarative code
    class$_2$ imperative and declarative code
    ...
    class$_n$ imperative and declarative code
END MODULE.

KNOWLEDGE Modules

KNOWLEDGE modules contain the initial state of the knowledge base for each class. A KNOWLEDGE module may not have IMPORT statements in it. KNOWLEDGE modules are written in the DOME declarative language and are interpretively executed at runtime. The KNOWLEDGE module has the form:

KNOWLEDGE MODULE *module_name;*
    class$_1$ knowledge base definition
    class$_2$ knowledge base definition
    ...
    class$_n$ knowledge base definition
END MODULE.

INTERFACE Modules

INTERFACE modules provide a mechanism to interface with existing code bodies. Currently the implementation supports C and C++, but it can readily be extended to other languages. Through this facility the programmer can "wrapper" existing C library functionality into an IMPORT class and make

use of these classes as if the implementation was in IMPORT. That is, there are no restrictions on inheritance and polymorphism.

INTERFACE MODULE *module_name;*
IMPORT *statements*
external_class$_1$ language specific renaming
external_class$_2$ language specific renaming
...
external_class$_n$ language specific renaming
END MODULE.

Keys

The key statement is used to specify applications. The key specifies an object to be created, and a method of that object that will be the entry point of the application. There may be any number of keys related to the IMPORT modules in a repository. Keys are stored in the repository with the corresponding modules. Keys are used by the interpreter and code generator to determine the modules and objects involved in an application.

The key statement may be put in a separate file, or within a file containing a DEFINITION MODULE or an IMPLEMENTATION MODULE. If it occurs within such a file, it must occur outside the MODULE-END MODULE block. A key must be given a unique name within the repository where the suite of modules resides. A key declaration has the form:

KEY *key_name*:
FROM *module_name* CREATE *class_name* INVOKE *method_name (arguments).*

Types

IMPORT uses name equivalence to determine types. The only nonatomic types that IMPORT allows are arrays, classes, and lists. Every expression has a statically-determined type.

Assignability and type compatibility is discussed in the individual subsections. Unless specifically allowed, types are generally not assignment compatible.

Ordinal Types

There are two ordinal types: enumerations and integers. An enumeration type is declared as follows:

TYPE T = {id_1, id_2, . . . , id_n};

In this case, id_1 will have an integer value of 0, id_2 a value of 1, and id_n a value of n-1.

The operators ORD and VAL convert between enumeration types and integers. In addition, the predicate ODD returns TRUE when the value of an ordinal is odd. INC and DEC increment and decrement the value of an ordinal respectively.

The identifiers id_1, id_2, etc. can be used to index arrays, and are compatible with integers in relational operations, but not arithmetic operations or otherwise.

There are two predefined enumerated types: BOOLEAN, which is the enumeration {TRUE, FALSE} and CHAR, which is an enumeration of 256 elements, generally the character set of the implementation. CHR is defined on integers and converts an integer into the appropriate character.

The elements of the enumerations must be distinct. Declarations of the form:

```
TYPE CITRUS-FRUITS = (ORANGE, GRAPEFRUIT, LEMON);
TYPE BREAKFAST = (MILK, GRAPEFRUIT, TOAST, EGGS);
```

are not allowed, since GRAPEFRUIT appears twice. If they are in different modules, however, they can be renamed to be distinct with the IMPORT statement.

Floating Point Types

At the present time, there is one floating point type REAL. The basic arithmetic and relational operators apply to REALs.

The following built-in functions apply to REAL data-type:

FLOAT (*i*)	Converts INTEGER to REAL.
TRUNC (*r*)	Truncates a REAL into an INTEGER.

Strings

STRINGS are defined as a first-class data-type in IMPORT. The STRING stores characters and is automatically and dynamically allocated and reallocated. STRINGs are indexed from 0 as are all arrays in IMPORT. However, STRINGs are not indexable via the [] operator as are arrays.

The following built-in functions apply to STRINGs:

CHARTOSTR (*c*)	Converts CHAR into STRING.
INTTOSTR (*i*)	Converts INTEGER into STRING.
LOWER (*str*)	Returns a STRING with all lower case letters.
POSITION (str_1, str_2)	Returns the position of str_2 in str_1.
REALTOSTR (*r*)	Converts REAL into STRING.
SCHAR (*str*, *pos*)	Returns the CHAR at position *pos* in *str*.
STRCAT (str_1, str_2, . . . , str_n)	Produces a new STRING which is a concatenation of all the STRINGs str_1, str_2, . . . , str_n.
STRLEN (*str*)	Returns the length of the STRING.
STRPUT (arg_1, arg_2, . . . , arg_n)	Converts all the arguments arg_1, arg_2, . . . , arg_n into STRINGS and concatenates them together.
STRTOINT (*str*)	Converts STRING into INTEGER.
STRTOREAL (*str*)	Converts STRING into REAL.

SUBSTR (pos_1, pos_2, *str*)	Returns the substring from pos_1 to pos_2.
UPPER (*str*)	Returns a STRING with all upper case letters.

The following built-in procedures apply to STRINGs:

INSERT (str_1, *pos*, str_2)	Inserts str_2 into str_1 at pos.
REPLACE (str_1, pos_1, pos_2, str_2)	Replaces the part of str_1 from pos_1 to pos_2 with str_2.
STRTOCHAR (*str*, *array_of_char*)	Converts a STRING into an ARRAY OF CHAR. The size of the array must be sufficient.

Arrays

An *array* is an indexed collection of elements of a given type. Indices to an array must be expressions of INTEGER type or an enumerated type. All array indices start at 0, and if the size of an array is *n* then the maximum addressable element is *n-1*. Multidimensional arrays are allowed.

Examples:

```
TYPE MY_ARRAY = ARRAY [20] OF INTEGER;
TYPE NAME_ARRAY = ARRAY [20] OF ARRAY OF [10] OF CHAR;
```

The first declaration is of a one-dimensional array of INTEGERs and the second a two-dimensional array of CHARs.

If an array A is of type T, then A[*i*], where *i* is an integer, is the *i*th element of A. Array names may not be used in comparisons or arithmetic operations.

Lists

The LIST type is supplied to provide for integration of imperative IMPORT methods and the DOME declarative methods. A *list* is an ordered collection of heterogeneous elements. The elements may be of any type. An empty list is denoted by NULL.

The following operations are defined on lists:

HEAD (*list*)	Returns the element at the head of the LIST.
TAIL (*list*)	Returns the LIST starting at the second element.
APPEND ($list_1$, $list_2$)	Creates a new LIST from $list_1$ and $list_2$ and returns it.
NTH (*int*, *list*)	Returns the *n*th element of the *list*.
LENGTH (*list*)	Returns the length of the *list*.

Objects

An object is either NULL or an instance of its *class*. The class of an object defines an object's *members*. There are three types of members: *data members* (or *fields*), *imperative methods*, and *declarative methods*. The field and methods of an object are defined in the DEFINITION module. The imperative methods are contained in the IMPLEMENTATION or INTERFACE modules, and the declarative methods (or knowledge-base) are contained in the KNOWLEDGE module.

The values of data members constitute the *state* of an object. The state of an object can only be changed by the methods of the object. A method is a body of imperative or declarative code, which may or may not alter the state of an object.

A declaration of a class in the DEFINITION module has the form:

TYPE *identifier* = OBJECT (*superclass*$_1$, *superclass*$_2$, . . . , *superclass*$_n$)
    field and method list
PRIVATE
    field and method list
OVERRIDE
    field and method list
END OBJECT;

Only the method *prototypes* are listed in the declaration. The actual methods are defined in the IMPLEMENTATION module. The IMPLEMENTATION module contains the method bodies for each class:

OBJECT *class*$_1$;
    method$_1$
    method$_2$
    . . .
    method$_m$
END OBJECT

In the case of the INTERFACE module, the language specific renaming of objects, fields, and methods is given for code generation compatibility purposes. The use of the extended syntax for renaming is illustrated below. The AS key word is used to specify the name of an external object, field, or method. The STATIC key word is used to indicate a C++ *static* or *class* member. AS and STATIC may be combined.

OBJECT *class*$_1$ AS *c++_class_name*$_1$;
    field$_1$ AS *c++_field_name*$_1$;
    method$_1$ AS *c++_method_name*$_1$;
    method$_2$ STATIC;
    . . .
    method$_m$ AS *c++ _method_name*$_m$ STATIC;
END OBJECT

Data Members. The fields of an object are declared as follows:

id$_1$, *id*$_2$, . . . , *id*$_n$: *type*;

Within each method, instance variables of the object may be accessed as if they were variables in the local scope. Data members of super classes may also be accessed in this manner. Beyond the scope of the class, methods of other instances may obtain a value of an instance variable through an ASK method invocation of the form:

ASK *obj_name* id_x;

which will return the value of id_x.

These methods are automatically defined when the data members of a class are defined. In addition, a member SELF is automatically defined for each class, and always contains the identity of the instance. This member may not be assigned to.

Methods. There are six types of methods: CONSTRUCTOR methods, DESTRUCTOR methods, ASK methods, TELL methods, QUERY methods, and OPERATOR methods. With exception of the DESTRUCTOR method, methods may be overloaded as long as *signatures* can be distinguished.

The arguments to these methods may be passed by value, or by reference. Each parameter must be declared either to be IN (arguments to be passed by value), OUT, or INOUT (arguments to be passed by reference).

CONSTRUCTOR methods are automatically invoked when an object instance is allocated. CONSTRUCTOR methods may be overloaded, but may not specify a return value. The CONSTRUCTOR method for an object is always named ObjInit. A CONSTRUCTOR method prototype looks like:

CONSTRUCTOR METHOD ObjInit (IN $param_1$);

DESTRUCTOR methods are automatically invoked when an object instance is deallocated. Only one DESTRUCTOR method may be defined per class, and it may have no arguments and may return no value. It is always named ObjTerminate. A DESTRUCTOR method prototype looks like:

DESTRUCTOR METHOD ObjTerminate();

ASK methods are synchronous bodies of imperative code and may be used in expressions. They may return a value, and may contain all three kinds of parameters: IN, OUT, and INOUT. An ASK method prototype looks like:

ASK METHOD *method_name* (IN $param_1$: INTEGER, INOUT $param_2$, $param_3$: REAL) : REAL;

The method defined takes three arguments and returns a REAL.

TELL methods are asynchronous bodies of imperative code: in the current implementation a separate light-weight process is created for them, and there is no guarantee on order of evaluation. As a result, they are not allowed to return any value, and may only have IN parameters. For more information on the interaction of methods, see *Simulation Environment*, p 32.

TELL METHOD *method_name* (IN $param_1$: REAL);

QUERY methods are synchronous, and may not advance simulation time. They serve as the interface to the Theorem prover, and are always BOOLEAN functions. There is no restriction on the types of parameters. For more information, see *The Theorem Prover*, p 26.

QUERY METHOD *method_name* (INOUT $param_1$) : BOOLEAN;

OPERATOR methods are the mechanism by which the programmer may overload arithmetic, relational, and assignment operators. They must be binary functions, taking two arguments of the same type: the class on which the OPERATOR is defined. It must also return an instance of the same class. The arguments are passed by value. The operator to be overloaded is given as the *method_name*.

OPERATOR METHOD + (IN $param_1$, $param_2$: *my_class*) : *my_class*;

Knowledge Base. The knowledge base body is a collection of expressions that is initially in each object's knowledge base when that object is allocated. This knowledge base is defined for each class and may be empty. The knowledge base is specified in a KNOWLEDGE MODULE and has the form:

OBJECT *class_name*;
    $expression_1$
    $expression_2$
    . . .
    $expression_n$
END OBJECT;

The expressions are DOME *Goal Expressions*, described in **Expressions**, p 24.

Inheritance. A class may have one or more *superclasses* from which it *inherits* members. If there are conflicting member names, such members cannot be accessed directly by an instance of a subclass and an error will be reported at compile time.

The fields and methods of a superclass, however, may be overridden with the OVERRIDE directive. Fields and methods in the override section of the class declaration supersede those of the superclasses.

An object may have *private* members not visible to methods outside the class. Private members are declared after the keyword PRIVATE.

If a variable v_1 is of class C_1, and variable v_2 is of class C_2 and C_1 is a superclass of C_2 then v_2 is assignable to v_1. However, v_1 is not assignable to v_2 since v_2 may have fields and methods not present in v_1.

To retrieve a conflicting field (present in both C_1 and C_2), assign v_2 to v_1, and invoke the appropriate method. *Method Calls* (p 25) details this process.

Within a method of a v_2, we may access a member of its superclass C_1 by using the INHERITED FROM construct:

INHERITED FROM C_1, f_1

where f_1 is a field defined in C_1. This is only necessary if f_1 has been overridden in C_1.

Declarations

IMPORT is a strongly typed language, and variables, constants, classes, and methods must be declared before being used. Class and method declarations are made in the DEFINITION modules, while variable declarations can only be found in the IMPLEMENTATION modules. Constants may be declared in both kinds of modules. Class and method declarations have already been described in **Types**, p 11.

Occasionally, recursive or interleaved class structures may be required, and a forward declaration may be necessary. In this case, the keyword FORWARD is used to mark the class.

```
TYPE my_obj = OBJECT; FORWARD;
TYPE not_my_obj = OBJECT;
    field_one : my_obj
END OBJECT;
```

Classes implemented in other languages are declared as shown above in the DEFINITION module, but with the additional EXTERN statement added. The default foreign language is C++. The INTERFACE module facility provides for renaming of classes, fields, and methods as might be necessary to match up with existing code. Additionally, they may be declared STATIC as IMPORT does not support the concept of *static* class members and functions as does C++.

```
TYPE external_obj = OBJECT; EXTERNAL;
field and method list
END OBJECT;
```

Constants

Constants may be defined in DEFINITION modules and at any place variable definitions are legal. These are at the start of new *scopes*: at the beginning of a method or BLOCK statement. Constant declarations have the form:

CONST id_1 = *expression*$_1$;
id_2 = *expression*$_2$;

All identifiers used in expressions defining constants must also be constants that have already been defined.

Variables

Variables may be declared at the start of methods and BLOCK statements. IMPORT is a lexically scoped language.

Variable definitions take the form:

VAR id_1 : *type*$_1$;
id_2, id_3 : *type*$_2$;

Method Bodies

Method bodies are the only places where imperative or declarative code can be found. Excepting the QUERY method, method bodies have the form:

method_prototype
local variable and constant declarations
BEGIN
statement_list
END METHOD.

QUERY methods do not have associated bodies in the IMPLEMENTATION module. Instead, they are entry points into the knowledge base for the class. When a QUERY method is invoked, it is translated into a new expression to be resolved by the Theorem Prover.

Statements

Statements within a method or BLOCK are executed sequentially. Statements are terminated with a semicolon.

Assignment

There are 5 forms of the assignment statement, corresponding to each of the four basic arithmetic operators and a simple assignment:

x:= 3;	assigns 3 to location x.
x+= 3;	increments x by 3.
x-= 3;	decrements x by 3.
x/= 2;	divides x by 2.
x*= 3 + 5;	multiplies x by 8.

The types of the location (the variable on the left of the assignment statement) must match the type of value generated by the expression (on the right). These operators may be overloaded with the OPERATOR method of a class.

Method Invocations

Method invocations are of the forms:

method_type object_name TO *method_name(argument_list)*;
method_type object_name ABOUT *method_name(argument_list)*;

The *object_name* is a variable that points to an object. The keywords TO (used for ASK and TELL methods), and ABOUT (used for QUERY methods) are noise words and are optional. ASK, TELL, and QUERY methods are invoked in this way. OPERATOR methods are invoked simply by using an arithmetic operator in the normal infix notation.

Examples:

```
ASK missile_1 TO fire_at(target_9);
TELL tank_commander TO attack(hill_291);
QUERY foreman ABOUT fits_in_schedule(project);

new_marble_set:= old_marble_set + winnings;
```

Here, the assignment operator is also overloaded.

There are predefined methods for each field of an instance. If *f* is a field of instance *v*, then:

ASK *v f*;

returns the value of *f*.

If *v* is of type C_2 which inherits from C_1, then we can invoke methods of C_1 by using the INHERITED FROM construct:

INHERITED FROM C_1 ASK *v method_name(arguments)*;

BLOCK Statements

BLOCK statements are a way of creating a new, local scope within a method. A BLOCK statement has the following form:

```
BLOCK
    local constant and variable declarations
BEGIN
    statement list
END BLOCK;
```

Variables declared within a BLOCK statement are only applicable within it, and within its subblocks.

IF Statements

An IF statement has the form:

```
IF expression1
    statement_list
ELSIF expression2
    statement_list
ELSIF expression3
    statement_list
    . . .
ELSE
    statement_list
END IF;
```

The ELSIF and ELSE clauses are optional and may be omitted. The expressions must be of type BOOLEAN. The expressions will be evaluated in order until one of them evaluates to be TRUE. Then the statements in the corresponding *statement_list* will be executed.

If no expression evaluates to TRUE, the ELSE clause will take effect. If there is no ELSE clause, the IF statement will have no effect except for side effects of the boolean expressions.

CASE Statement

CASE statements are a more efficient way of comparing an expression with a set of constants, when there is more than one execution path. The constant can be a predefined CONST of the same type as the expression, or more usually members of an enumeration type.

A CASE statement has the form:

```
CASE expression
WHEN const1:
    statement_list
WHEN const2:
    statement_list
    . . .
OTHERWISE
    statement_list
END CASE;
```

The OTHERWISE clause is optional.

Loop Control Statements

There are four ways to write loops in IMPORT. In each of these forms, the loop terminates if an EXIT statement is reached.

```
LOOP
```

```
LOOP
    statement_list
END LOOP;
```

This loop must contain an EXIT statement in order to terminate.

```
WHILE
```

```
WHILE expression
    statement_list
END WHILE;
```

The *expression* must evaluate to a BOOLEAN type. If *expression* evaluates to TRUE, the *statement_list* will be executed once, before *expression* is evaluated again.

REPEAT

REPEAT
    statement_list
UNTIL *expression*;

The *expression* is of type BOOLEAN. *statement_list* will be executed once, then if *expression* evaluates to TRUE, the loop will terminate.

FOR

FOR *id*:= *first* TO *last* BY *step*
    statement_list
END FOR;

id is a variable of an ordinal type. *first*, *last*, and *step* are expressions of the same ordinal type as *id*. At the first iteration of the loop, *id* is set to *first*, and thereafter it is incremented by *step*. The loop terminates when the value of *id* is greater or equal to *last*.

The keyword TO may be replaced by DOWNTO, in which case, *id* is decremented by *step*, and the loop terminates when the value of *id* is less than or equal to *last*. At the termination of the loop, *id* holds the last value used in checking the termination condition.

RETURN Statement

RETURN *expression*;

The RETURN statement is used to restore control to an invoking method. The type of *expression* must match the return type of the method in which the RETURN statement is found.

TERMINATE Statement

TERMINATE;

The terminate statement is used to stop the natural sequence of execution within a TELL method. It is recursive; the termination of the current method also terminates the method that invoked it.

WAIT Statement

WAIT statements are used in ASK and TELL methods to pass simulation time. They may have an INTERRUPT clause which will be executed if the method is interrupted with the INTERRUPT call imported from the run-time simulation library. More information on Interrupt can be found in *Simulation Support*, p 32.

WAIT DURATION *expression*
    statement_list
ON INTERRUPT
    statement_list
END WAIT;

There are three forms of the WAIT statement. In the first, *expression* is of type INTEGER and gives the amount of time to be passed in the simulation.

```
WAIT FOR object_name TO method_name(arguments)
    statement_list
ON INTERRUPT
    statement_list
END WAIT;
```

In the second, the time passed is specified within the method that is invoked, and the *statement_list* is executed after the method *method_name* is invoked.

```
WAIT FOR TRIGGER trigger_obj
    statement_list
ON INTERRUPT
    statement_list
END WAIT;
```

In the third, the method will wait for the specified Trigger object to fire. Trigger objects fire when their Fire method is invoked. *Simulation Support*, p 32 provides more information on the Trigger object.

INTERRUPT Statement

There are two built-in procedures supplied for stopping TELL methods that are currently executing within an instance of an object. They are the INTERRUPT and INTERRUPTALL:

INTERRUPT (*instance*, *method_name*)
Interrupts the first TELL method of *instance* whose name is stored as *string*.
INTERRUPTALL (*instance*)
Interrupts all TELL methods of *instance*.
INTERRUPTALL (*instance*, *method_name*)
Interrupts all TELL methods of *instance* with the given string literal name.

TRANSACTION and ABORT Statements

The concept of a transaction is basic to shared database systems. Transactions are necessary to permit multi-user concurrent access to shared data in a database. For applications to be able to read and update the same data consistently, protocols have been developed to overcome problems such as deadlock that arbitrate access to shared data. A transaction is a sequence of program statements that has exclusive control over some shared data. Once an application has achieved control of a portion of a database, other applications must wait for it to finish before they have access to that data. Thus transactions permit a group of program actions on data to occur without interruption. This is necessary to guarantee integrity of the database system. Note that transaction boundaries within concurrent database applications must be planned wisely. The goal is to permit other applications access to data also. The amount of data locked within a transaction is a prime consideration in multi-user database applications; such data must be structured appropriately.

A major design requirement for IMPORT is to provide multi-user concurrent access to shared databases. Therefore it is necessary to provide some form of transaction management flexibility within the language. In IMPORT, this is accomplished with the TRANSACTION statement. The TRANSACTION and ABORT statements are used when interacting with persistent storage. Any nontransient NEW

statement, that is, one that allocates into secondary storage, must be enclosed by a TRANSACTION statement. This is true for any statement that modifies the state of a Database. The actions on a Database within a transaction are atomic, and either all actions on a Database within a transaction are completed, or none are completed.

A transaction may be aborted by the object-oriented database system, or it may be aborted by the programmer with the ABORT statement. TRANSACTION statements may have an ABORT clause, which is executed when the transaction is aborted. A TRANSACTION statement has the form:

```
TRANSACTION
    statement_list
ON ABORT
    statement_list
END TRANSACTION;
```

Abort statements have two forms:

```
ABORT;
ABORT ALL;
```

The first form aborts the deepest enclosing TRANSACTION, and ABORT ALL aborts all enclosing transactions.

IMPORT Statements

IMPORT statements are used to share classes between modules, and have the form:

FROM $module_1$ IMPORT id_1, id_2;
FROM $module_2$ IMPORT ALL id_3;
FROM $module_3$ IMPORT id_4 AS id_5;
FROM $module_4$ IMPORT id_6(id_7 AS id_8);

The first statement imports id_1 and id_2. The second statement imports id_3, which is an enumerated type, and all its members. The third statement imports id_4 in $module_3$ and renames it to id_5 in this module. The fourth statement imports id_6, an enumerated type, and only one member of the enumeration id_7, and renames it to id_8.

Dynamic Memory Management

NEW

The NEW built-in procedure allocates memory for the creation of object instances. Suppose id_1 is of type $class_1$. Then NEW (id_1) creates an instance of $class_1$ in transient memory. Additionally, parameters to the CONSTRUCTOR method of an object may be specified:

NEW (id_1(arg_1, . . . , arg_n)

To allocate an object instance into persistent store,

NEW (id_1, *database_in_use*)

creates a persistent instance of $class_1$ in the object-oriented database *database_in_use*. *database_in_use* is a variable of type Database. Objects can also be allocated in Segments and Configurations. The class libraries used to support persistence are covered briefly in *Database Class Library*, p 32 and fully described in Appendix B.

DISPOSE

The DISPOSE built-in procedure returns transient objects to the heap and removes a persistent object from the database:

DISPOSE(*my_obj*);

Expressions

An expression is a computation that produces a value. An expression is either an operand or an operator applied to operands. An expression is evaluated by recursively evaluating its operands, then applying the operator to it. The order of argument evaluation is undefined; in particular, there are no "short-circuit" BOOLEAN expressions.

Operators and Precedence

Method invocations, built-in function calls and array indexing always have the highest priority, after which the priority and associativity are listed in decreasing priority (Table 1). All the operators are infix operators.

Integer Literals

INTEGER literals may be represented in decimal or hexadecimal (when appended by the letter H).

Example:

A091H

Table 1

Operator Associativity by Decreasing Priority

Operator	Associativity
unary –	right
*/DIV MOD	left
+-	left
= <> < > >= <=	non-associative
NOT	right
AND	left
OR	left

Character Literals

Character literals are represented by a printable character in single quotes or a decimal number from 0 to 255 followed by the letter C.

Example:

```
54C
```

String Literals

String literals consist of a sequence of printable characters on one line within double quotes.

Example:

```
"This is a string."
```

Floating Point Literals

Floating point REAL literals are represented in decimal form, or in scientific notation.

Examples:

```
1.0
0.31
1.3E+20
```

All literals must have an integer portion and a fraction portion. The numbers: 1. and .2, are not legal floating point literals.

Lists

Lists are denoted by expressions separated by commas within square brackets.

Example:

```
["string", 34, y]
```

The example shows a list of three elements, consisting of a string, an integer, and the value of variable y.

Designators

A designator is an expression that denotes the location where a value is held. Variables used in expressions are designators. $a[i]$, where a is an array, and i is an ordinal, denotes the ith element in array a, and is also a designator. These are the only two kinds of designators.

Method Calls

Invocations of methods that return a value are also expressions. The type of the expression is the type of the return value of the method.

Arithmetic Operations

The basic arithmetic operations are built into the language and are defined for REALs and INTEGERs. In addition, DIV, which returns the quotient of an INTEGER division, and MOD, which returns the remainder of an INTEGER division, are also defined. All arithmetic operators are infix operators. The type of the expression returned by the arithmetic operation is the type of its operands.

Relations

The relations less than (<), greater than (>), equal to (=), less than or equal to (<=), greater than or equal to (>=) and not equal to (<>) are built into the language for ordinals and floating point numbers. The type of a relation is always BOOLEAN.

Boolean Operations

The operations logical-and (AND), logical-or (OR), and logical-not (NOT) are defined for BOOLEANs. The expression formed with these operators are also of type BOOLEAN.

NULL

The keyword NULL designates a null value. This can be an object, or a STRING.

UNINSTANTIATED

The keyword UNINSTANTIATED, when assigned to a variable, denotes a variable that has no value. This is used in conjunction with QUERY methods to determine the result of computation. For more information, see **The Theorem Prover** (p 26).

The value of UNINSTANTIATED is implementation dependent, but is usually the largest negative INTEGER for integer and enumerated types, 0 for CHARs, NaN (not a number) for floating points, and a pointer to a unique location, usually NULL, for other types of variables.

Relationship to DOME

IMPORT was designed to support a declarative programming facility. This capability is supplied through integration with DOME, which was developed in conjunction with IMPORT (Whitehurst and Pietrzak 1993).

The DOME Theorem Prover

The DOME theorem prover is based on Prolog, with a similar Horn Clause unification model. Each expression represents a Horn Clause, and is asserted into the knowledge base of the object that contains it. Clauses can be added to a knowledge base through the KNOWLEDGE section of an object, or through QUERY method definitions.

Within a proof, all clauses are considered equivalent. The difference between the two ways of adding clauses to an object involve how a proof is invoked: All clauses that appear in the KNOWLEDGE section are for the internal use of the theorem prover. QUERY methods, however, can be called from other IMPORT methods, in a manner similar to ASK and TELL method invocations.

The order of clauses with a head of the same name is significant; during a proof, the *i*th clause encountered will be chosen before the *i*th + 1.

Parameters within a QUERY invocation are split logically into two types: constant (or instantiated), and uninstantiated. The constant parameters can be any constant or variable declared within IMPORT, except those variables specifically declared as uninstantiated. Constant parameters are used directly by the theorem prover, and are not modified. Uninstantiated variables are modified depending on the success of the proof.

When a QUERY method is invoked, the theorem prover will attempt to unify the given parameters with the clauses available in that object's knowledge base. Due to inheritance, it may also search other knowledge bases for a solution. When it has finished attempting to prove the query, the theorem prover will return true or false depending on its success. If it returns true, it will also assign the substituted values used to solve the proof into the corresponding uninstantiated variables passed on (Figure 2).

Note: Only the first solution found by the theorem prover will be returned; other potential solutions are ignored.

DOME Goal Expressions

The DOME "goal" expression is encoded into an assertion or a query to be executed by the DOME proof procedure. These expressions can only be found in the KNOWLEDGE module. The syntax of a DOME goal expression is always of the form:

$$f(t_1, t_2, \ldots, t_n)$$

where f is the name of the goal, and the t_is are the arguments of the goal. Each t_is is a DOME term.

An example of such an expression:

age (john, 23)

where "john" is a symbol, and "23" is an integer constant. Both are considered as terms in DOME parlance.

DOME Terms. There are three kinds of DOME terms: constants, variables, and lists. A constant DOME term can be an integer, string, symbol, etc., defined just as normal constants are within IMPORT. However, variables within IMPORT are treated as either constant DOME terms or variable DOME terms, depending on their value: An instantiated variable is considered to be constant by the theorem prover, but, an uninstantiated variable is considered to be a variable DOME term when given in a query, and thus the theorem prover is allowed to assign some value to it. A list is a variable of type LIST.

Success	Constant	Uninstantiated
FALSE	No effect	No effect
TRUE	No effect	Instantiated, if necessary

Figure 2. Effect of Return of QUERY Invocation On Given Parameters.

Built-in Procedures and Functions

IMPORT inherits the built-in procedures and functions of ModSim with a few additions, all of which are reproduced here for easy reference. The built-in procedures are shown below using a syntax close to that of methods in terms of their parameters. In the following descriptions, square brackets, [], represent optional arguments.

Built-in Procedures

ABORT [ALL]

Aborts the current transaction. ABORT ALL aborts the current and all enclosing transaction.

DEC(INOUT *arg* : *AnyOrdinal* [; *n* : INTEGER])

Decrements any ordinal (enumeration, integer, or character) typed variable. Decrements by one unless optional parameter specifies differently.

DISPOSE(IN *arg* : *AnyObject*)

Free memory associated with a variable of type OBJECT: transient or persistent.

HALT

Terminates execution of a program. All transactions are aborted.

INC(INOUT *arg* : *AnyOrdinal* [; *n* : INTEGER])

Increments any ordinal (enumeration, integer, or character) typed variable. Increments by one unless optional parameter specifies differently.

INPUT(OUT arg_1 : *SomeType* [;arg_2 : *SomeType* . . .])

Reads from standard input and places values into each of the parameters. Input values may be separated by spaces, tabs, or newlines. *SomeType* must be CHAR, INTEGER, REAL, or STRING.

INSERT(INOUT str_1 : STRING; IN *pos* : INTEGER; IN str_1 : STRING)

Inserts str_1 at *pos* in str_2. Numbering of positions in STRINGs begins at position zero (0).

NEW(OUT *obj* : *AnyObject* [; IN *Database* | *Segment* | *Configuration*])

Allocates storage to a variable, *obj*, in either transient or persistent memory. Persistent allocation requires specification of a location in terms of an object of type *Database*, *Segment*, or *Configuration*.

OUTPUT(IN arg_1 : *SomeType* [; arg_2 : *SomeType* . . .])

Places the contents of the variables given as parameters on the standard output device. *SomeType* must be CHAR, INTEGER, REAL, or STRING.

REPLACE(INOUT str_1 : STRING; IN pos_1, pos_2 : INTEGER; IN str_2 : STRING)

Replaces that part of str_1 beginning at pos_1 and ending at pos_2 with str_2.

STRTOCHAR(IN *str* : STRING; OUT *ArrayOfChar* : ARRAY OF CHAR)

Converts *str* to an : ARRAY OF CHAR) stored in the variable *ArrayOfChar*.

INTERRUPT(IN *obj* : *AnyObject*; IN *method* : STRING)

Interrupts most imminent *method* of *AnyObject* returning control to the point of invocation, activating the ON INTERRUPT clause, if any.

INTERRUPTALL(IN *obj* : *AnyObject* [;IN *method* : STRING)

Interrupts the **ALL** methods named *method* of *AnyObject*. If no method is specified, interrupts all pending TELL methods.

Built-in Functions

ABS(IN *arg* : INTEGER | REAL) : INTEGER | REAL

Returns the absolute value of the argument.

APPEND(IN $list_1$, $list_2$, LIST) : LIST

Creates a new LIST from $list_1$ and $list_2$ and returns it.

CAP(IN *ch* : CHAR) : CHAR

Returns the capital (uppercase) character corresponding to the argument.

CHARTOSTR(IN *ArrayOfChar* : ARRAY OF CHAR) : STRING

Returns a value of type STRING based on conversion of the input ARRAY OF CHAR type argument.

CHR(IN *n* : INTEGER) : CHAR

Returns the CHAR corresponding to the INTEGER argument. The range is 0-255.

FLOAT(IN *n* : INTEGER) : REAL

Returns the REAL value of the INTEGER argument.

HEAD(IN *list* : LIST) : LIST

Returns the element at the head of the LIST *list*.

INTTOSTR(IN *n* : INTEGER) : STRING

Returns a STRING value representing the INTEGER argument.

LENGTH(IN *list* : LIST) : INTEGER

Returns the length of the *list*.

LOWER(IN *str* : STRING) : STRING

Returns a value of type STRING corresponding to the argument in all lowercase.

MAX(*ScalarType*) : ScalarType

Returns the maximum value of the given type that is representable by the computer.

MAXOF(IN arg_1 : *ScalarType* [;IN arg_1 : *ScalarType*]) : *ScalarType*

Returns the maximum value of all the arguments. The arguments must be of the same type.

MIN(*ScalarType*) : ScalarType

Returns the mimimum value of the given type that is representable by the computer.

MINOF(IN arg_1 : *ScalarType* [;IN arg_1 : *ScalarType*]) : *ScalarType*

Returns the minimum value of all the arguments. The arguments must be of the same type.

NTH(IN *int* : INTEGER; IN *list*) : LIST

Returns the *n*th element of the *list*.

ODD(IN *n* : INTEGER) : BOOLEAN

Returns TRUE if the argment is odd, otherwise FALSE.

ORD(IN *arg* : OrdinalType) : INTEGER

Returns the ordinal value of the argument.

PERSISTENT(IN *obj* : ObjectType) : BOOLEAN

Returns TRUE if the argument has been allocated to persistent store, otherwise FALSE.

POSITION(IN str_1, str_2 : STRING) : INTEGER

Returns the position of str_1 in str_2. Position is numbered begining from zero (0).

REALTOSTR(IN *arg* : REAL): STRING

Returns a STRING value representing the REAL argument.

ROUND(IN *arg* : REAL) : REAL

Returns the argument rounded to the nearest integer.

SCHAR(IN *str* : STRING; IN *pos* : INTEGER) : CHAR

Returns the character at position *pos* counting from zero (0) as the first position.

SIMTIME() : INTEGER

Returns the current simulation time.

STRCAT(IN str_1, str_2 [, str_n]) : STRING

Returns the STRING consisting of the concatenation of the arguments.

STRLEN(IN *str* : STRING) : INTEGER

Returns the length of the STRING *str* as an INTEGER.

STRPUT(IN arg_1 : *SomeType* [; arg_2 : *SomeType* . . .]) : STRING

Returns the STRING consisting of the concatenation of the arguments. The arguments are converted to their STRING representation. *SomeType* . . . may be CHAR, INTEGER, REAL, ARRAY OF CHAR, or STRING.

STRTOINT(IN *str* : STRING) : INTEGER

Returns the INTEGER representation of the STRING argument.

STRTOREAL(IN *str* : STRING) : REAL

Returns the REAL representation of the STRING argument.

SUBSTR(IN pos_1, pos_2 : INTEGER; IN *str* : STRING) : STRING

Returns the part of the *str* beginning at pos_1 and ending at pos_2. Positions are numbered beginning at zero (0).

TAIL(IN *list* : LIST) : LIST

Returns the LIST starting at the second element.

TRUNC(IN *arg* : REAL) : INTEGER

Returns the INTEGER valued part of the REAL parameter.

UPPER(IN *str* : STRING) : STRING

Returns a value of type STRING corresponding to the argument in all uppercase.

VAL(IN *OrdinalTypeName*; IN *OrdNum* : INTEGER) : *OrdinalType*

Returns the value of the ordinal type name, e.g., CHAR, at the given ordinal position.

Standard Library Modules

The IMPORT language depends on *Standard Libraries* to achieve certain capabilities. At present, these consist of modules for simulation support and persistence. These are needed as a minimum, and additional modules are planned to provide a basis for general applications development.

Simulation Support

Simulation time is kept as an INTEGER with no associated units.

SIMTIME ()

Returns an INTEGER that is the current simulated time.

Simulation granularity is user-defined. Each TELL method creates a new task and any TELL method may create more than one active task at a time.

The main feature of the simulation environment beyond the WAIT and TELL constructs, is the Trigger class, used for synchronization. The runtime module is included automatically with each module and contains the definition of the Trigger.

```
DEFINITION MODULE RT;

    Trigger = OBJECT;
    ASK METHOD Fire ();
    ASK METHOD InterruptTrigger ();
    END OBJECT;

END MODULE.
```

Invoking the Fire () method of a Trigger releases all tasks that are waiting on the Trigger. Invoking InterruptTrigger () causes all tasks waiting on the Trigger to execute the ON INTERRUPT portion of the WAIT FOR TRIGGER statement instead.

Database Class Library

IMPORT provides a database class library supporting persistent object storage. Objects are allocated into persistent store using the NEW procedure (described in **Dynamic Memory Management**, p 23). All interaction with a database must be enclosed within a TRANSACTION statement, p 22. Table 2 lists the database classes. The full class definitions are given in Appendix B.

Table 2

Database Classes

Class	Description
Database	Contains information on where other objects are stored
Directory	Allows hierarchical storage of roots in the Database
List	An ordered collection of objects
ListCursor	Cursor for a List
Tree	Generalized tree of objects
TreeCursor	Cursor for a Tree
Configuration	Configuration for versioning
Workspace	Workspace, where configurations of objects are checked into, and modified

3 IMPLEMENTATION

The object-oriented approach to software design and development is most generally characterized as *modeling*. The resulting software systems can be thought of as *simulations*. IMPORT was motivated by the requirements for complex system modeling and simulation. To respond to these requirements, it was natural to take this same approach to implementation. In this regard, the implementation of IMPORT more closely resembles a state-of-the-art CAD system than a programming language. Dart (1990) gives an insightful analysis of the benefits achievable in software development environments by adopting CAD system approaches.

Introduction

The implementation strategy of IMPORT supports the goals of software engineering in general and anticipates specific requirements for its intended use within a software development environment. From the software engineering perspective, IMPORT is implemented as an object-oriented framework modeling an *Intermediate Representation* of the language. The modeling approach supports *understandability* and *maintainability* as it reflects a close correlation between conventional complier constructs and software classes. The tools chosen for implementation are *reliable*, *efficient*, and *portable*. The framework approach is the best current thinking in software engineering for *reusability* through design iteration (Johnson and Foote 1988). The *verification* of a language is more constrained than many general applications, and test suites have been developed that exercise all specifications of the implementation. *Validation* of the requirements on which the design rests requires the use of the language in the development of applications of the type for which it was built. Efforts in this direction are underway.

Requirements differentiating IMPORT from other languages are those related to software development environments. Generally, IMPORT is intended to be used within a distributed, collaborative software engineering development environment. The trend toward the use of object-oriented database systems as the underlying mechanism to support such systems seems clear. This approach permits a number of significant benefits and offers a more integrated solution to some longstanding problems. Perhaps the most interesting, from a research point of view, is in configuration management (Kalathil and Herring 1993). IMPORT may be the first fully functional programming language with an implementation based on an object-oriented database with the aim to provide consistent and integrated support within a software engineering environment. Longer term plans include knowledge-based software engineering that places further requirements on representations for reasoning about software artifacts (Lubars and Havandi 1987). Chapter 4 develops an overview of the motivation for these and other related requirements. Brown (1991) gives a good summary statement of requirements for software artifact object repositories.

This chapter describes the implementation of the IMPORT compiler, beginning with a brief overview of compilation as it relates to the structure of the IMPORT compiler (Aho, Sethi, and Ullman 1986).

Compilation is the process of transforming programs from the source language (high-level) format into some target (lower-level) format. This process can be divided into two stages: analysis and synthesis. The analysis phase decomposes source input into elemental components represented in some convenient intermediate form. The synthesis phase assembles the target format program based on the intermediate represenation. Analysis consists of three parts: linear, hierarchical, and semantic analysis. Linear analysis, often called lexical analysis, views the source program as a sequence of characters. During this phase, characters are read in and grouped into symbols or tokens of the language, i.e., keywords and identifiers. The tokens recognized in lexical analysis are passed on to the next step. In hierarchical analysis, often called syntax analysis or parsing, the tokens are further grouped into collections having some higher-level

meaning in the source language. These collections usually take the form of parse trees or syntax trees. That is, they are naturally hierarchical in structure. These hierarchical structures are processed during semantic analysis to ensure program correctness (at the next higher level of abstraction) and insert information needed for the subsequent synthesis phase. Synthesis uses the results of analysis to produce the target formated output. There are several approaches depending on the host and target hardware and software environment. It is common in the synthesis phase to produce another intermediate (machine independent) representation and to optimize it. This optimized form is then translated to the binary format of the target machine. System utilities take care of linking appropriate libraries and other needed runtime support to produce the final executable. Figure 3 gives an overview of the components of the IMPORT compiler.

The **Lexer**, **Parser**, and **Semantic Controller** constitute the components of the analysis phase. The bottom three, **Code Generator**, **Compiler/Linker**, and **Runtime** comprise the components of the synthesis phase. In traditional compilers, these components create intermediate representations in transient memory. IMPORT is distinct in that an **Object Repository** is used to store an extended intermediate representation. The repository is shown at the center of the diagram as all components interact with it in performing their respective actions. Necessary to all phases of the compilation process is the management of the **Symbol Table** and other bookkeeping functions.

Given this overview of the components of the compiler and their interaction, the presentation is organized as follows. Beginning with the object model of the intermediate representation, which forms the structure of the object repository and provides the basis for a discussion of the analysis portion. The synthesis part of the compiler follows, which includes the code generator, the DOME interpreter interface, and the simulation runtime support. Finally, the role played by the Generic Object-Oriented Database (GOOD) interface in implementation of the compiler and support for persistent programming in the IMPORT language is described.

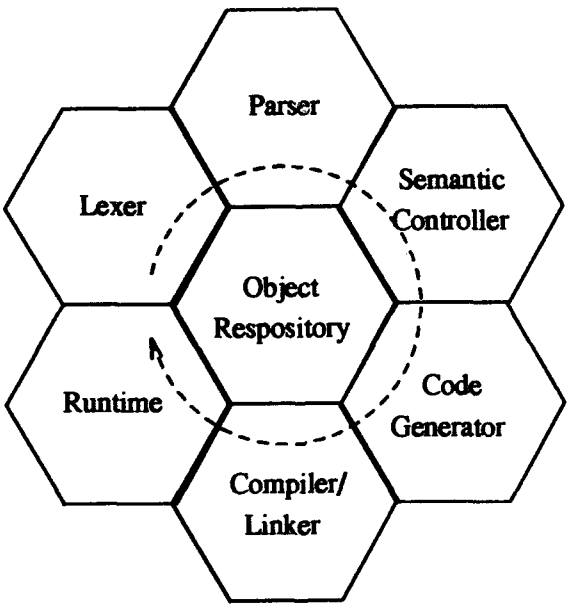

Figure 3. IMPORT Components and Their Relationships.

Modeling IMPORT: The Intermediate Representation

This section presents the intermediate representation designed to support compilation as well as the application domain goals of the IMPORT compiler. The representation is designed to be generic, and yet have enough correspondence with other modular object-oriented programming languages to facilitate translation to and from these languages. This especially helps in code generation. Appendix C includes the C++ header files for the classes described in this section.

The simplest internal representation of a program is a parse tree with an associated symbol table. However, additional information that is useful during compilation and in the development of software engineering tools is also required. To meet these goals, a set of Software Engineering Environment (SEE) classes that form the basis for interaction between the tool set were designed. (The overall structure of the environment and tools are discussed in Chapter 4.) The SEE classes are used to represent the imperative portion of an IMPORT program. The declarative portion of the program is translated into a knowledge-base representation. This takes the form of a collection of expressions identical to the implementation of the first class data-type LIST. Details of how this is used in the interface between IMPORT applications and the DOME theorem prover can be found in **The DOME Runtime Interpreter Interface**, p 46.

The general structure of a module (or application) is decomposed into the hierarchical form shown in Figure 4. This form follows the structure of the IMPORT language closely. A module contains declarations (constants and types) and classes. Classes are composed of declarations (fields) and methods. Methods contain local declarations (constants, types, and variables) and the statements that make up the body. During compilation, each node shown in the tree structure becomes an instance of one of the SEE classes.

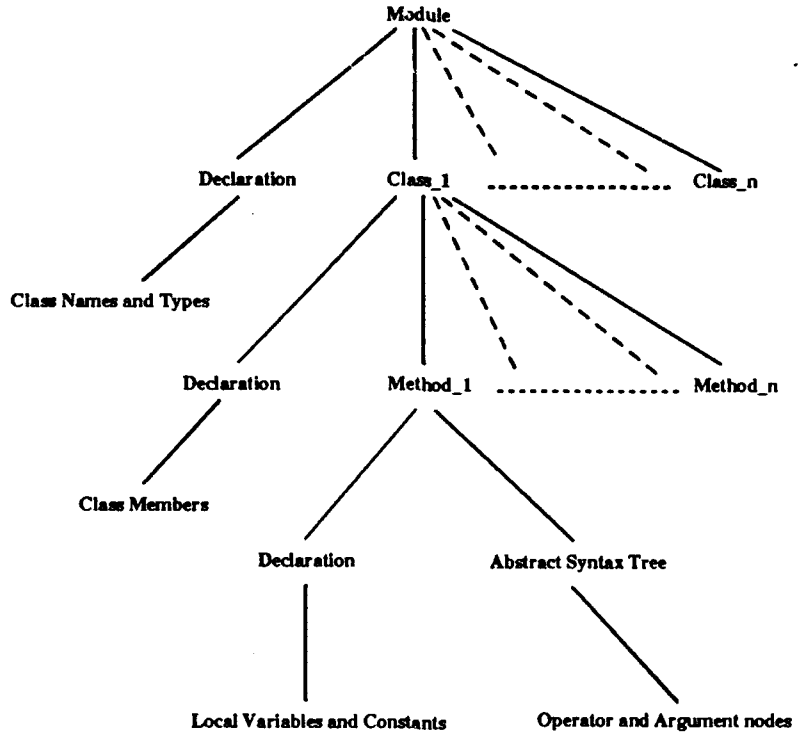

Figure 4. Modeling Software With Objects.

At this point, a more formal notation is introduced to describe the SEE classes. This notation comes from the object-oriented analysis and design methodology known as Object Modeling Technique (OMT) (Rumbaugh 1987). Explanations of the symbology used in OMT will be given as it is encountered. A limited subset of OMT was taken from the OMT *Object Model*. Figure 5 shows the OMT equivalent of Figure 4.

The "part-of" relationship among classes is called *aggregation*. The diagram shows the aggregation relationships among the SEE classes id_Module, id_Class, id_Method, Declaration, and ast_node. The diamond symbol (under id_Module) stands for the aggregation relationship that exists between id_Module and id_Class as explained above. The soild circle shows this relationship is "one-to-many": a module contains zero or more classes. A detailed presentation of structure of the SEE classes and their supporting classes is given below.

Modules

Each application is composed of modules. Figure 6 shows the attributes for the class id_Module. The id_Module attributes include the module_name, which is a string, and a module_type, which is an integer. Two attributes assisting in code generation are last_touch and has_external_obj. The first of these stores an integer value corresponding to the date and time of compilation, and the last indicates whether the module has externally defined classes. Three of the attributes are pointers to other classes. The decl is a pointer to an instance of class Declaration that is examined in detail later. The two remaining attributes are instances of *id_oodb_list* (See **The GOOD Common Interface**, p 53). The classes attribute is a list or collection of the classes declared (or defined) in the module, and imported_modules is a list of the modules imported for type visibility. An OMT class diagram shows methods of the class as well as attributes. In the case of id_Module, there is only one constructor and a destructor. These methods are omitted from this and future diagrams because they are assumed for all classes.

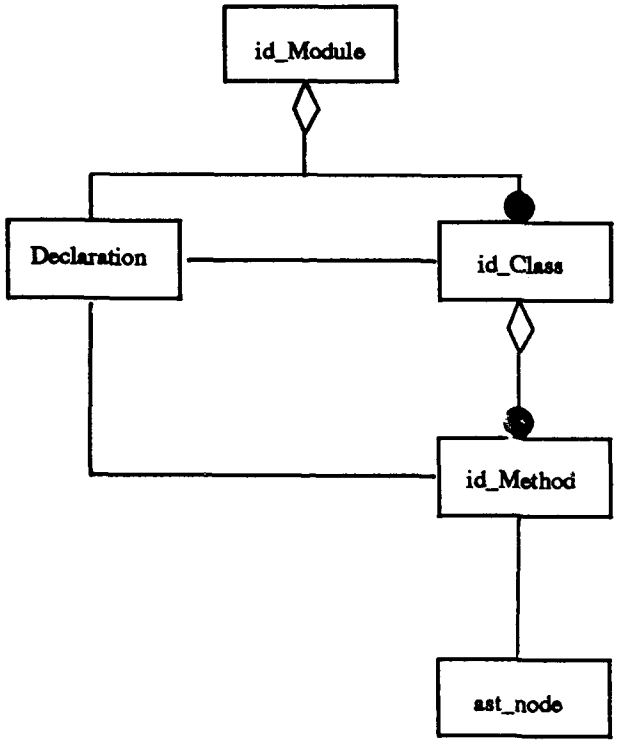

Figure 5. OMT Diagram of SEE Classes.

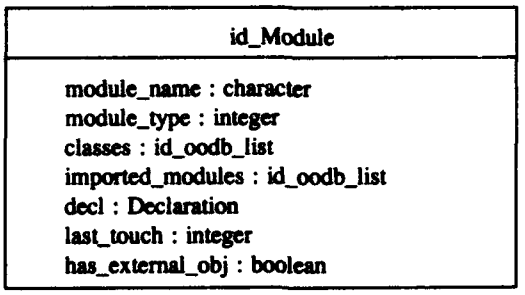

Figure 6. Class Diagram of the Id_Module Class.

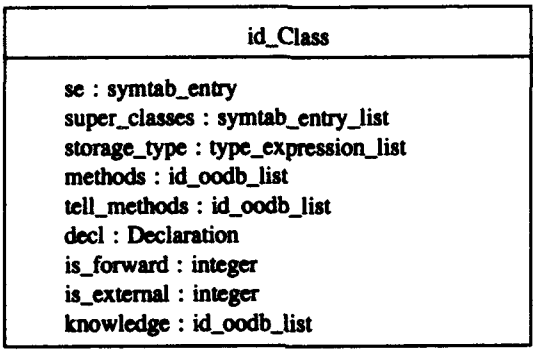

Figure 7. Class Diagram of id_Class.

Classes

The id_Class models an IMPORT "Object." It contains types and names of fields, access rights, and inheritance information. In each is also a collection of methods (Figure 7).

Each object in a module must be stored in the symbol table. id_Class contains an attribute for this purpose: *se*, which is a pointer to an instance of symtab_entry class. The attribute symtab_entry_list contains pointers to instances of symtab_entry for each class from which the instance inherits. This is necessary for code generation. Also, id_Class must store the methods of the class. This is accomplished by an id_oodb_list. A distinction is made for TELL methods as they require special handling during code generation. A pointer to an instance of Declaration, *decl*, holds the declaration of the fields of the object. is_forward and is_external are needed both for parsing and code generation. The declarative statements from the KNOWLEDGE module are stored in another id_oodb_list in the attribute gknow.

Methods

IMPORT methods are modeled by the id_Method class. Instances of this class describe the various methods supported by IMPORT, characteristics such as parameter names and types, the type of any value returned, local variables, and abstract syntax trees that describe the statements in each method. The id_Method class is shown below in Figure 8.

Each instance of id_Method contains a symbol table entry, *se*, for the method and a method type indicator. The parameters to the method, parms, are stored in an instance of symtab_entry_list and their types, parm_type, are stored in an instance of type_expression_list. The storage type of the returned value, if any, is found in storage_type of which is also represented as a type_expression_list. A pointer to an instance of Declaration, *decl*, stores the locally declared variables of the method and *asf* is a pointer to

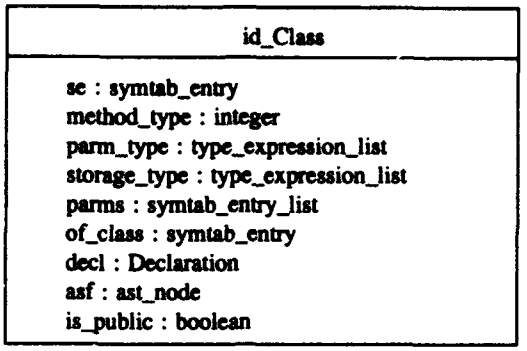

Figure 8. Class Diagram of id_Method.

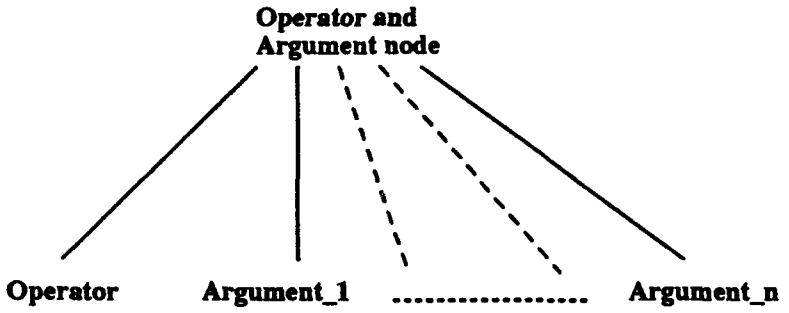

Figure 9. Node in the Abstract Syntax Tree.

the ast_node, which stores the body of the method. The is_private boolean attribute indicates if the method is declared PRIVATE in the definition module.

Abstract Syntax Nodes and Tree

The abstract syntax tree (*ast*) is a commonly used structure for storing information obtained about a program during parsing and for subsequent manipulations. The ast structure stores the minimal amount of information necessary (unlike parse trees). Each abstract syntax tree is composed of operator and argument (operand) nodes (Figure 9). In an ast, the operator is found at an interior node and the operands are the children, or leaves, of the operator. Attributes determined during parsing are stored in the nodes. The ast structures are built up during parsing to represent the statements in the program. In the case of IMPORT, they store statements found in method bodies. These sequences of statements take the form of ordered lists of ast nodes.

Each operator and argument node contains an operator and an ordered collection of operands, where each operand is an operator and argument node. Figure 10 shows the class diagram for the ast_node. The op attribute stores the particular operator of the ast node. These are the arithmetic, relational, boolean, keywords, built-ins, etc., of the language definition. The method and obj_instance attributes are only used (valid) for certain method invocations. The which_type attribute is a pointer to an instance of type_expression_list and stores the type of the operator. The final attribute is a union structure. This structure holds the semantic value or meaning of the operand. This includes the basic types, symbol table entry, or a list of operands for the node. There are constructors for the class that create each of the possible different ast node types.

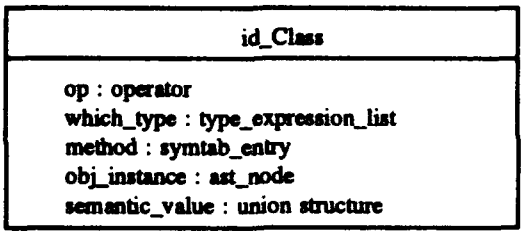

Figure 10. Class Diagram of ast_node.

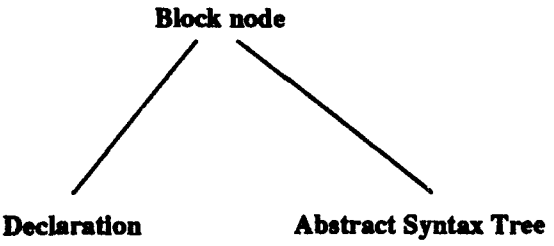

Figure 11. A Block Node With Declaration Object.

There are other useful attributes that can be included in the ast_node class to support the development of software tools. They may keep pointers to sections in the source, or the source itself at this level. This is to help with applications like structure editors and debuggers. Relevant sections of code can be accessed in this way. Other bookkeeping information can be stored at this level for such tools as run-time profilers.

The ast approach is used to implement the BLOCK statement in IMPORT (Figure 11). IMPORT is a lexically scoped language, and within a method or block, the BLOCK construct permits the definition of a new scope. Hence, the block node must keep track of the new environment in which the contained statements must be executed. This is accomplished by placing a Declaration object as the first operand in the operand list of the ast_node.

Supporting Classes

A number of classes are needed to support the SEE classes in compilation of IMPORT modules, code generation, and development of software tools. These classes have, necessarily, been referred to above in conjunction with the presentation of the SEE classes.

Type Expression and Type Expression List. The IMPORT compiler must perform static type checking of the source code modules. IMPORT is a strongly typed language and static type checking helps ensure program correctness. Static checking ensures that types produced by language constructs match. A type expression is the type, as defined by the language's type system, produced by some language construct. In IMPORT, there is a set of basic supplied types, e.g., INTEGER, and user-defined types, e.g., OBJECT. Many other constructs have type expression associated with them, for example, methods that return a value and built-in functions. Figure 12 shows the type_expression class.

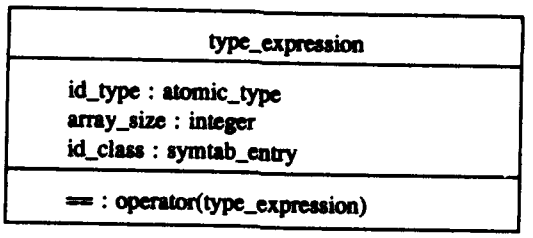

Figure 12. The type_expression Class.

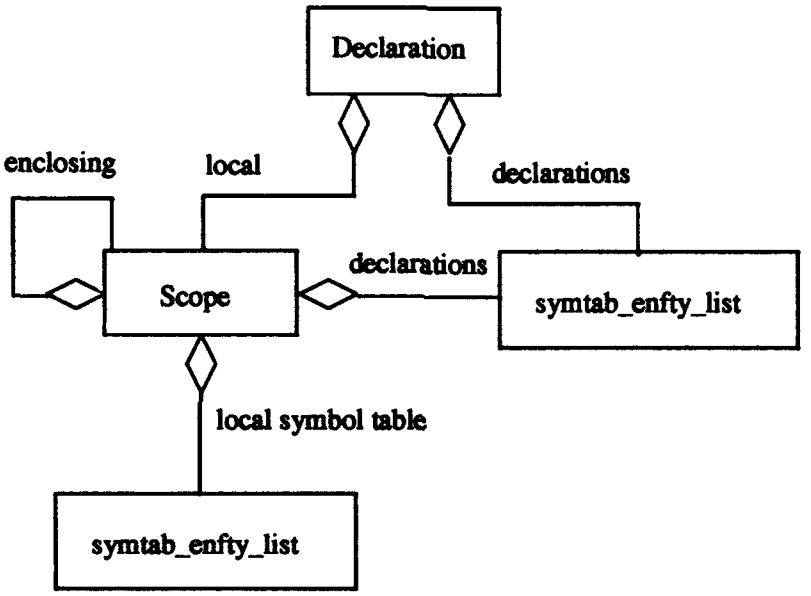

Figure 13. The Declaration and Scope Classes.

This simple class contains attributes for storing the type of a language construct. The attribute id_type stores the basic language defined types such as BOOLEAN, INTEGER, etc. If the type is OBJECT, then the id_class attribute refers to its symtab_entry. The array_size attribute stores the size of a type declared as an ARRAY is valid only for this type. In addition to constructor and destructor methods, type_expression has an operator method for testing equality.

The type_expression_list inherits from id_oodb_list and stores a collection of type_expression object instances. It has methods for operating on type_expression_list such as comparing, copying, and appending. It is used, for example, to pass method argument lists.

Declarations and Scopes. Declarations, such as of variables in a programming language, associate information with the name of the declared object. This information is used within the context of the scope rules of the language to determine if and when a name is valid and how it can be manipulated. The section of code (program text) in which a declaration is valid is called a scope. IMPORT is a block structure, lexically scoped language. During compilation, a symbol table entry is created in the symbol table on encountering a new declaration within a scope.

The Declaration class contains pointers to the smallest enclosing scope: the local scope. This scope, with all its ancestors, which represent other enclosing scopes, constitute the current environment for a lexically scoped language. The Declaration class contains an attribute that is a reference to a symtab_entry_list. The object referred to here is the set of declarations, stored in the symbol table, that are within the local scope. (This structure is necessary to run an interpreter using the intermediate representation of the program.)

The Scope class is used to manage declarations within a scope. It contains a reference to the previous enclosing scope and the depth of the current scope within the context of the environment. It has a list of declarations within the scope, and a local symbol table is provided for their access. The local symbol table is implemented as an open (no limit) hash table of entries, with one symbol table entry per identifier. These entries contain all the semantic and type information for that identifier. Figure 13 shows the relation of the Declaration class to the Scope and symtab_entry_list classes.

The Symbol Table and Symbol Table Entries. A symbol table is used to manage declarations and their scopes as they are encountered in modules and for code generation. The symbol table is organized as a directed acyclic graph of scopes, whose root (there being only one top level node) is the global environment. Immediately below this level is a collection of scopes for each individual programming module. Each entry in the symbol table, actually in the appropriate scope, corresponds to the name of a declaration. Access to declarations is efficiently implemented by hashing on the spelling (characters making up the name) of the declaration. The Symtab class is shown in Figure 14. Its relation to the Scope and symtab_entry class is shown in Figure 15.

Finally, the symtab_entry class contains the information associated with a named declaration. It stores the name, the generated name (for code generation), the symbol type, and other information (Figure 16). It also constrains references to instances of id_Class, id_Method, and type_expression_list. These are needed for access to complex symbol table entries such as objects, methods, type names, and variables.

In addition, the symbol table also keeps track of a "use list" for each variable (stored in a symtab_entry instance). The list keeps track of the blocks or sections of the parse tree where a variable is used. This is a service provided to support other tools, like structure editors and interpreters that can use this information to help the programmer maintain consistency when a variable declaration is changed.

Lexer, Parser, and Semantic Controller

The presentation of the SEE classes in the previous section explained the Object Model supporting both the analysis and synthesis phases of the IMPORT compiler. This section describes the tools used to construct the compiler, and other classes developed for the analysis phase, and relates them to the SEE classes. As described earlier, the analysis portion of the compiler consists of the lexer, parser, and semantic controller. Figure 17 shows a schematic of the components of the analysis phase.

The lexer and parser for the IMPORT compiler was built using Compiler Resources' Yacc++ (Revision 1.4). Yacc++ is an advanced, object-oriented, language construction tool written in C++. The objects in dashed boxes in Figure 17 are components from the Yacc++ *Language Objects Library* (LOL). The "lexer" is more than the usual deterministic finite automata (DFA) lexer, and is specified in terms of productions much as the parser is specified. This allows nested comments that would otherwise not be possible in a standard DFA. The parser produced by Yacc++ is a table-driven LR(1) parser superior to the commonly available Unix yacc LALR implementation. The accompanying reference manual fully describes the LOL classes (Compiler Resources 1992).

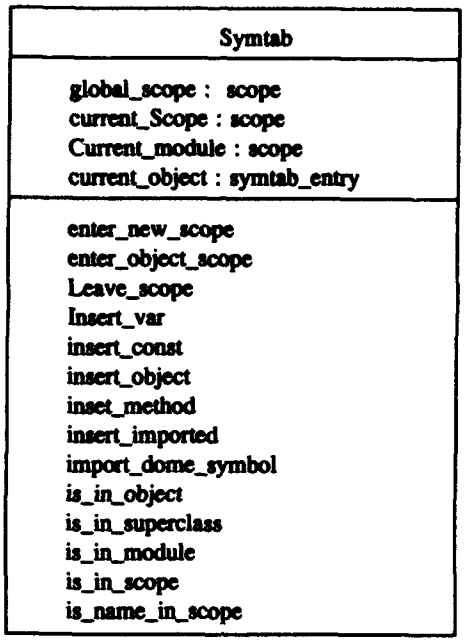

Figure 14. The *Symtab* Class.

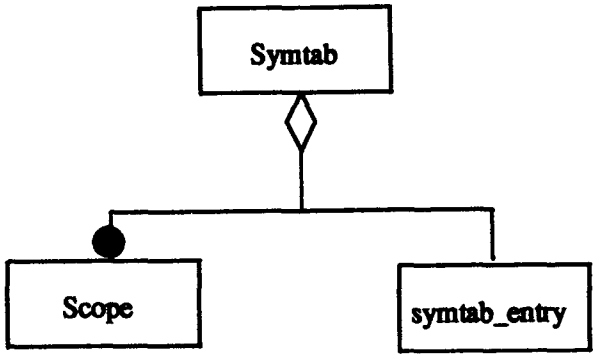

Figure 15. Relation of Symtab, Scope, and symtab_entry Classes.

This section describes the id_semantic_controller class and its relation to the SEE classes and the object repository. The semantic action code and the SEE classes are written in C++. The header files for these classes are listed in Appendix C. The semantics actions determined during parsing result in method invocations on an instance of id_semantic_controller. This object uses the SEE and associated support classes to construct the intermediate representation of the module being compiled. The intermediate representation is stored in an object repository created by use of the GOOD interface (described in **Generic Object-Oriented Database Interface**, p 52).

Figure 18 shows the id_semantic_controller class. It contains attributes, mostly pointers to object instances, determined by the state of the parser. It maintains references to the current module, object, and method for which it is constructing the intermediate represenation. Most of the methods of this class are of the form id_semantic_controller :: build_X, where X is a construct of the language. There is a method of the class for all constructs. These are called by the parser when it determines the construct.

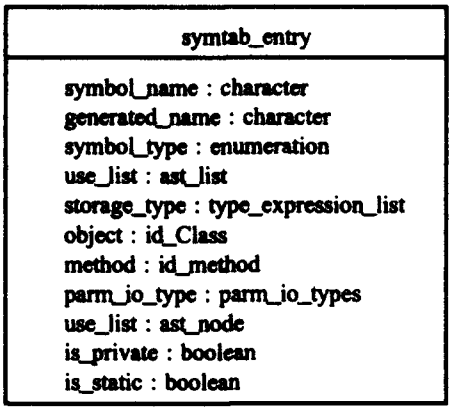

Figure 16. The symtab_entry Class.

Not brought out in the discussion of the SEE classes is the appearance of the id_oodb_list class as attributes. This class is heavily used in the implementation as it provides support for storage of lists of persistent objects. This is one of the classes of the GOOD interface. Several of the SEE classes inherit from id_oodb_list. These are ast_list, type_expression_list, and symtab_entry_list. It is through the use of these classes and persistent object allocation functions of the GOOD that id_semantic_controller builds the persistent intermediate representation.

A separate lexer and parser was also created for DOME to produce the intermediate form of the declarative code found in KNOWLEDGE modules. This parser was constructed with the same tools as the IMPORT parser described above. Yacc++ provides for the collaboration of these two distinct parsers within a single application.

Code Generator, Compiling, and Linking

It was decided that the code generator should produce C++ code for a number of reasons. Since it has replaced C as the de facto systems programming language, there is a close mapping between the constructs found in IMPORT and C++. This greatly facilitates code generation. Features such as inheritance, and operator and method overloading, for example, are much easier to achieve. This significantly reduced the amount of effort needed to quickly get the language running, especially when compared to generating machine instructions directly. This approach should also provide for cross-platform portability.

The code generator produces C++ code from the intermediate representation stored in the object repository. It takes as input commands the specification of a path to the repository and the name of a module or a key. The class id_code_generator performs the reverse operation of id_semantic_controller of the last section. It has basically the same attributes and an id_code_generator :: generate_X method where X corresponds to a languge construct. This class performs a pre-order traversal of the intermediate representation to produce the corresponding C++ code for each ast node. It is, however, specific to IMPORT since it requires knowledge of the persistent class libraries, and the IMPORT runtime libraries to generate the code required by each of them. It also produces the C++ statements to create the structures required by the theorem prover at runtime.

Compiling and linking is performed in the usual manner. The software engineering environment tools to support the process of generating the correct version of an application as well as assisting in compiling of IMPORT programs is described in Chapter 4.

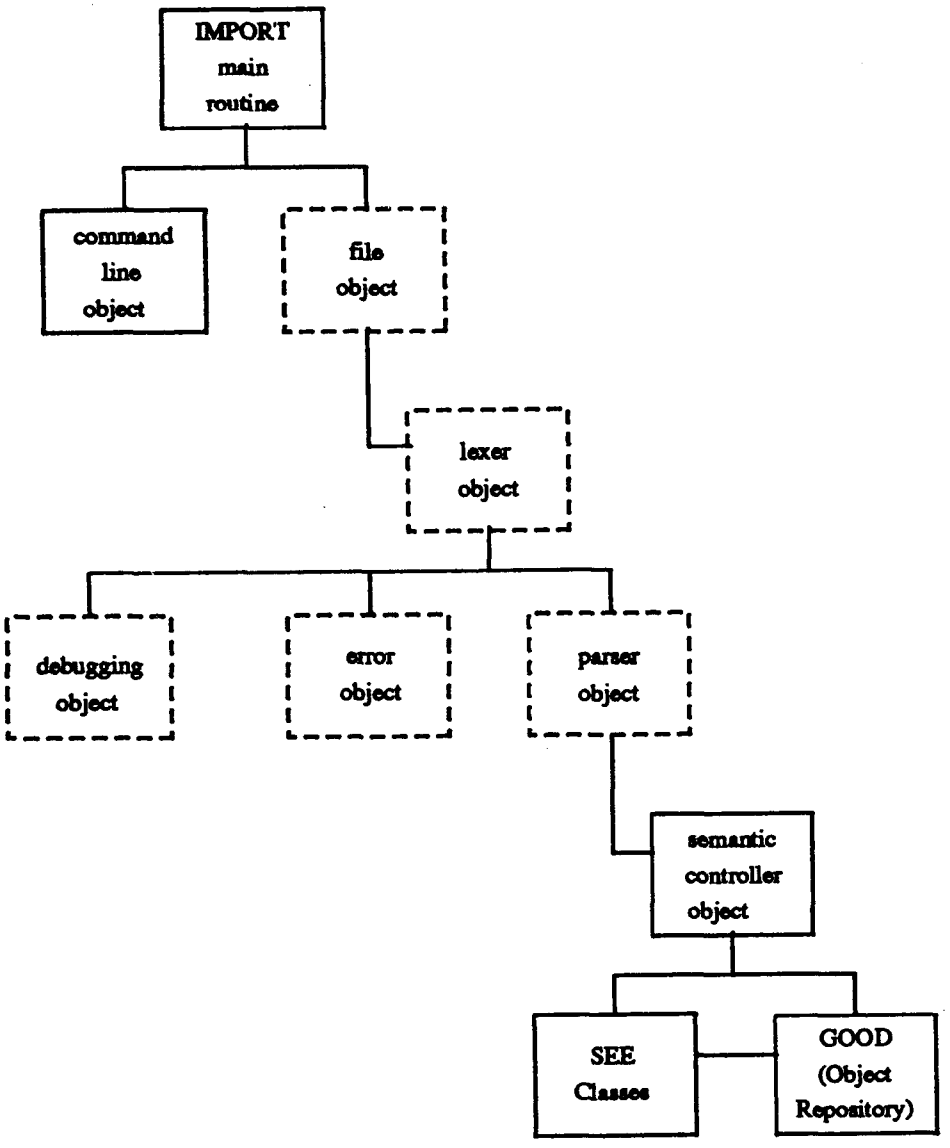

Figure 17. Components of the Analysis Phase.

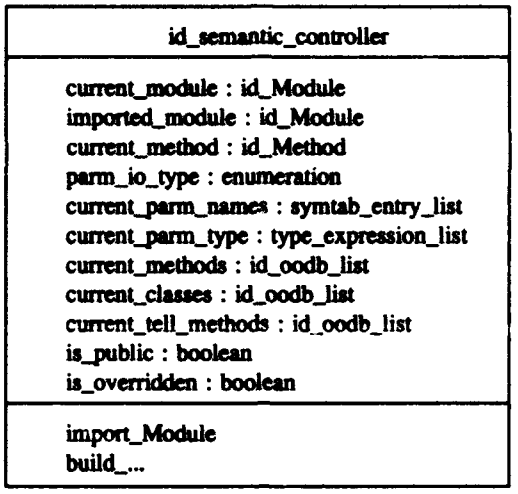

Figure 18. The id_semantic_controller Class.

The DOME Runtime Interpreter Interface

A goal of IMPORT/DOME is to provide the ability to make design trade-off decisions between the efficiency of imperative code, and the intuitiveness of declarative code. The integration between the two programming styles is achieved through: shared data-types and data structures. This is done by having a common symbol table, runtime type-checking, and direct manipulation of objects in the object repository by the DOME theorem prover.

Launching a Query

The DOME theorem prover is started whenever a QUERY method of an object is invoked. At compile time, each QUERY method invocation is translated into a call to the theorem prover with the appropriate expression denoted by the method and its parameters. An expression structure that can be used by the theorem prover is built at runtime into the object repository or a runtime database. The theorem prover then attempts to unify the expression with the knowledge base of the object whose method was invoked.

QUERY methods must have a mechanism to permit them to communicate values found in the imperative (compiled) code to the knowledge base. They must also return values from the knowledge base to variables located in imperative methods. This is done by passing parameters as arguments to QUERY methods. If a parameter is designated as an INOUT parameter, the theorem prover can assign a value to it (when it successfully unifies the QUERY expression). This permits the value to be assigned to a variable in the imperative code.

Shared Data-Types and Data Structures

To reduce data impedance between the theorem prover and the compiled code, two data-types (enumerated types and LISTs) are shared. These type structures may be accessed by both the compiled code and the theorem prover. Other first class data-types: INTEGERs, REALs, STRINGs, and CHARs, can also be passed as arguments to the theorem prover. This necessitates an UNINSTANTIATED value for all variables, so that they may contain the results of unification returned by the theorem prover.

The use of a persistent object repository or a runtime database allows the theorem prover to be disassociated with the compiled code, so it can run as a separate process, and interact with compiled code only through the persistent database and perhaps an inter-process socket. Several applications could be served in this manner, or perhaps several different interpreters with different unification strategies could be used by a single application.

Enumerated Types. Traditionally, symbols in an interpreted Prolog-like language, such as DOME, have no semantics beyond the fact that they can be used to satisfy some unification rule. In IMPORT/DOME, all symbols that may be returned or used in imperative code have to be declared as part of an enumerated type. Hence, each symbol is associated with a type and an ordinal value. These semantics are very important since the result of a unification can now be used in comparisons or as array indices in imperative code.

Lists. LISTs are basic structures in the theorem prover. All expressions that the theorem prover manipulates are LISTs. LISTs are first-class data-types in IMPORT/DOME and may be passed as parameters to QUERY methods and received as results of a unification.

Runtime Type-Checking. Since the symbol table from the imperative code is stored in a object repository, runtime type information is available to the theorem prover. This means that after a symbol has been unified, the theorem prover can use the symbol table to find the relevant type information and check to see if it is of the type expected for the parameter of the QUERY method. If there is no type conflict, then a successful unification is assumed, and the semantic value of the symbol is returned.

Direct Manipulation of Object

The theorem prover should go beyond reasoning about the fixed set of clauses in the knowledge base of the local object (the object whose QUERY method launched the computation). It should be able to provide different solutions based on some current state of the world. This state would presumably be provided by the data members of the local object and some other objects "known" to the local object. Since these objects can be stored in a database, they are accessible to the theorem prover, which can translate their state into some form suitable for unification.

This aspect of the language is still under investigation, and it has not been determined how we can "introduce" one object to another so that the first object may use the second object's state in the reasoning process. Current proposals include the use of automatically created class extents, from which an object may find a relevant instance that should be included in its knowledge.

Simulation Runtime Support

This section describes the runtime system for the IMPORT language, which is a sequential implementation for single processor environments. The runtime libraries and scheduler were developed using the AT&T task classes and collections classes from the *tools.h++* commercial package produced by Rogue Wave Software (1992). Rogue Wave allows the binary libraries to be distributed as part of another application without restriction.

Semantics of Simulation Constructs

The design of the runtime system is motivated by the requirements of the simulation time passing and synchronization constructs of IMPORT. The semantics of each of these constructs is described below.

SIMTIME. SIMTIME returns the current time according to a user-defined simulated scale.

TELL Method Invocations. Each TELL method invocation results in the creation of a new thread of control, which semantically is assumed to exist concurrently (in simulated time) with the activation that made the invocation. Furthermore, this new thread of control may or may not become *synchronized* with some other thread of control executing in the system, depending on the nature of the invocation mechanism. Thus, the runtime system must provide support for creation of an *asynchronous* thread of control (which executes the code body of the TELL method), and allow for subsequent synchronization of this thread. The synchronization constructs provided in the IMPORT language take the form described below.

WAIT DURATION Statements. Simulation time elapses by the execution WAIT DURATION statements, which cause the current thread of control to yield control to another ready thread and to resume execution either when the specified duration of simulated time has elapsed, or if the wait is interrupted. The language allows different actions to be specified depending on the cause of the resumption. Thus, the runtime system needs to provide support for suspending executing threads, and resuming at different points in the code.

WAIT FOR (TELL Method Invocation). This is one of the synchronization constructs provided in the language that allows the invoking thread to wait for the termination of the invoked thread. A thread is assumed to have terminated when its execution encounters a return from the corresponding method body. The language allows the wait to be interrupted by the following constructs, which requires the runtime system to provide support similar to that required for the WAIT DURATION statement. A record of the activity (waiting for a duration of simulation time to elapse, or for another TELL method to terminate) needs to be kept that associates the waiting thread with the activity it is waiting upon. Since both WAIT FOR and WAIT DURATION statements can also appear in ASK methods, the notion of a thread needs to be well defined so as to record all activities happening as part of the same thread at the same place.

WAIT FOR TRIGGER To Fire. This is another synchronization construct used to synchronize a group of threads. All threads wait on a special runtime object called the Trigger and resume operation when another thread executes the Fire method of the Trigger object. Support similar to the previous statement needs to be provided in the runtime system.

INTERRUPT and INTERRUPTALL. The language provides constructs for asynchronously resuming a thread of control that is either waiting for a duration of simulated time to elapse, or is waiting for the termination of another thread. The interrupted thread is specified by the TELL method name associated with the thread, and the object instance on which the method has been invoked. This requires the runtime system to categorize the different thread activities by the type of the TELL method. The semantics of the INTERRUPT constructs specify that the thread to be resumed is the one that is likely to resume the earliest, provided its wait is not interrupted. Thus, some notion of when an activity is likely to complete should also be present in the activity records.

TERMINATE. A thread terminates itself when it executes the TERMINATE statement. The runtime system needs to ensure that when a thread terminates, the TERMINATE is propagated recursively to all threads that are waiting on this thread.

Details of the Implementation

The runtime system has been built using the AT&T *task library*, which provides support for the management of lightweight threads of control. Figure 19 shows the overall structure of the runtime system.

The encapsulated region shows the functionality that the AT&T tasking package provides: it takes care of scheduling threads that are not waiting on other activities. The suspension and resumption of threads that are effected by switching the contexts of two threads are all provided by the task library.

A thread gets created whenever a TELL method is invoked. The different activities correspond to the execution of WAIT DURATION and WAIT FOR statements. The main components of the implementation are discussed below.

Activity Records. To ensure uniform treatment of both WAIT DURATION and WAIT FOR activities, the implementation treats the activity corresponding to WAIT DURATION also as being an activity that is waiting for a thread to terminate. This effect is achieved by creating a Timer thread that terminates after waiting for the specified duration of simulation time to elapse. Thus, all activities initiated from threads created as a result of user-specified TELL methods wait for the termination of either Timer threads, or other user threads.

Each activity record has three fields: an identifier for the thread requesting the activity, the thread being waited upon, and an estimate of the time when the activity is likely to complete. It is possible to accurately record the termination time for WAIT DURATION activities; however, WAIT FOR activities do not lend themselves to a similar situation. The implementation sets the estimated time for termination of WAIT FOR activities to be set to the time of initiation of the activity.

Activity Table. Whenever an activity is initiated, an activity record gets stored in the Activity Table corresponding to the thread that initiated the activity. Activity Tables associate all the activities corresponding to threads created by TELL method invocations on a particular object. These activities are categorized by the name of the TELL method that creates the thread. Furthermore, the activities are ordered with respect to the estimate of their termination times.

Activity Tables are queried whenever the activities corresponding to a TELL method of a particular object are interrupted via the INTERRUPT or INTERRUPT-ALL statements.

Task–Activity Table Associations. It is important to understand what it means to interrupt an activity corresponding to a TELL method invocation on a particular object. TELL method invocations create threads, which in turn initiate activities whenever WAIT DURATION and WAIT FOR statements are executed. Note that these statements could be encountered in the body of the TELL method whose invocation created the thread, or any ASK method that was invoked from this TELL method. A distinction needs to be made between the Activity Table of the object whose method is being currently executed, and the Activity Table of the object whose TELL method invocation created the thread. All activities initiated by a thread should be stored in the Activity Table corresponding to the latter. Since the information about the parent class of the TELL method is lost once we exit the scope of the TELL method body, information about the proper Activity Table to record all activities in must somehow be inferred given the thread identifier. The *Task-Activity Table Associations* record this relationship.

List of Garbage Tasks. Reclamation of storage allocated to threads and their entries in the various tables is an important issue. Storage corresponding to threads created as a result of WAIT FOR activities is very easy to reclaim: the storage is returned to the heap when control returns to the initiating thread.

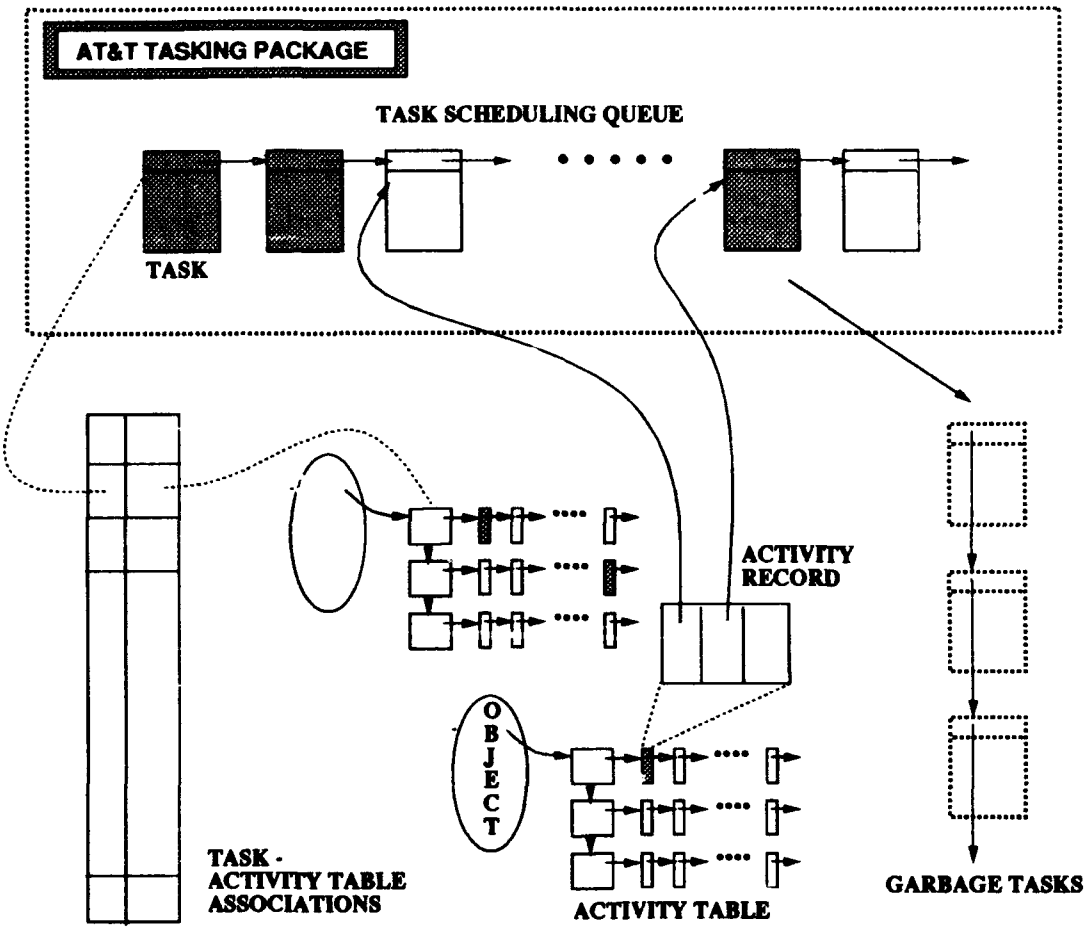

Figure 19. Overall Structure of the Runtime System.

However, asynchronous invocations of TELL methods pose a problem, since there is no way of detecting when these invocations terminate. The implementation supports a primitive garbage-collection facility, which consists of: (1) recording the creation of a thread due to an asynchronous invocation of a TELL method, (2) moving of the thread to a list of garbage tasks whenever the thread terminates (done by the thread itself), and (3) releasing the storage associated with terminated tasks, whenever the garbage task list is found not to be empty.

The association tables, and the sorted list of activities stored in the Activity Table have been implemented using the classes provided by the Rogue Wave Class Library.

The main protocols used in the implementation are described below. A thread starts executing under the control of the AT&T *task library* whenever a class derived from the base class `task` is instantiated. A thread waits on other threads by invoking a method of class `task, resultis ()`. This causes the initiating thread to get stored in the list of pending threads at the initiated thread. When the initiated thread terminates, all pending threads are activated.

Initiating an Activity. An activity record is created and inserted into the Activity Table corresponding to the initiating thread. The initiating thread then synchronizes with the initiated thread

using the `resultis ()` method. The status of the `resultis ()` call can be used to determine whether a particular activity completed, was interrupted, or encountered a TERMINATE statement. In all cases, the storage associated with the initiated thread is returned to heap after control returns to the initiating thread.

Terminating a Thread. When a thread executes a TERMINATE statement, it causes all calls to `resultis ()` for this thread to return a status indicating that it was terminated. This status has the effect of recursively terminating all threads that were pending on a terminated thread.

Interrupting an Activity. The semantics of the INTERRUPT and INTERRUPT-ALL statement call for interrupting an activity corresponding to a specified TELL method invocation of a particular object. The activity to be interrupted is obtained by looking up the Activity Table and picking up the most imminent activity categorized under the specified TELL method. If the activity corresponds to waiting on a Timer thread, then it can be interrupted outright, and the space reclaimed. However, if the activity corresponds to waiting for the termination of another user thread, the semantics of INTERRUPT need to be recursively propagated to this thread. Only when all threads being awaited for have been interrupted, can the space corresponding to a particular thread be reclaimed.

The implementation provides all the runtime functionality through a set of macros that are called in the generated code.

Code Generation

In addition to the use of the runtime macros whenever the IMPORT statements discussed in this section are encountered, the following changes are required to generate code that can use the AT&T task library:

- Each TELL method prototype is changed to return `(task *)` instead of `void`. The returned thread pointer is used in the runtime macros to perform synchronizations on threads.

- The body of each TELL method is moved from the original method definition in the enclosing class to the constructor of a new " TELL method class" which is derived from class `task`. This is the essential interface with the AT&T task library that is required to create a lightweight task whenever a TELL method is invoked. The original TELL method definition is changed to a creation statement for the "TELL method class."

- Each "TELL method class" constructor is enclosed in a `METHOD_PROLOGUE . . . EXIT_METHOD` body, which takes care of setting up the association tables and reclaims space on exit.

Extensions

The most important extension to the runtime system that has not been currently addressed is the integration of real-time control with the simulation system. This can take the form of a graphical user interface that is controlling the overall simulation system, which includes the user as one of its components, and which might inject events into the system.

The AT&T task library provides support for real-time signal handling as part of the overall simulation structure. It is conceivable that this be extended to meet the requirements of the above task. The overall scheme (which needs refinement) for doing this would be to define a signal which models the interface with the external world, and to define an `interruptHandler` task that contains the code for

what ought to happen when the signal does come through. User program threads need to wait on this `interruptHandler` task after they have finished whatever was initiated by the previous event that was injected into the system. The difficulty is two-fold:

1. Deciding when the previous event has finished processing, and causing all the user tasks to wait upon this `interruptHandler`.

2. Ensuring that the processing in the simulation system suspends while waiting for the signal to come from the outside world. This is related to the previous point since nonwaiting threads should not be allowed to proceed ahead in simulated time.

Generic Object-Oriented Database Interface

Object-Oriented Database Management Systems (OODBMS) were developed to support application areas that need to model complex data such as computer-aided design and manufacturing, and software engineering. The IMPORT/DOME language system integrates object-oriented procedural and declarative programming, simulation, and persistent programming. IMPORT is intended to be part of an integrated software engineering environment. Therefore, it must provide for the storage and manipulation software objects such as source programs, class definitions, libraries, compiled code, and intermediate representations of the compiled code in a consistent manner to simplify the development of software support tools. The availability of viable OODBMS, in a large sense, is responsible for, and made possible by IMPORT/DOME. Its implementation is predicated on OODBMS technology.

Both commercial and public domain OODBMS were investigated for use as the underlying database of IMPORT. From the beginning, it was decided not to tie IMPORT to any particular existing OODBMS. Users of IMPORT with access to a commercial OODBMS may want to use it as the underlying database for performance or other reasons. On the other hand, users without access to an expensive commercial OODBMS may want to use a public domain system.

Considering the above factors, IMPORT was designed with a common OODBMS interface that provides the necessary functionality. The interface was built using C++ and consists of classes that provide database functionality such as *database, directory, set, list, tree, set-cursor, list-cursor, tree-cursor, configuration, workspace*, and *segment.*

We have already hinted at the need for a common interface to the underlying OODBMS. The need for a common interface to the underlying OODBMS of the system is explained in the next section. Following that is a description of the database classes that constitute the Generic Object-Oriented Database (GOOD) interface. This chapter concludes with a summary of the current status of and plans for future work in this area.

Need for a Common Interface

One motivation for building a common interface to the underlying OODBMS of the IMPORT system is that different users may need to use different underlying databases. While some users may want to use a commercial database because of performance or other considerations, other users may want to use a public domain database for cost considerations. By not tying the OODBMS interface to a specific database system, the system can make use of new and improved OODBMS systems that may be available in the future.

Another motivation for providing a common interface is the need to support persistence with the IMPORT language itself. This approach permits the use of the common interface as the mechanism to achieve persistence in IMPORT in a consistent manner, and to provide for portabilitiy as well.

The common interface provides a mechanism that allows the differences between the underlying database implementations to be hidden from the user. Features that are in the common interface but not in the underlying database can be built on top of the database. Also, the common interface can be made simpler than that of the underlying database by assuming away some of the details of the underlying database system.

Several commercial systems were reviewed: ObjectStore (Object Design 1991), the ARPA OpenDB (Wells, Blakely, and Thompson 1992), the ITASCA distributed object database management system (ITASCA Systems 1991), and a public domain database, the Object System of STONE (OBST) (Forschungszentrum Informatik 1992). The *ObjectStore* database system is a powerful commercial OODBMS that provides three interfaces to the database, namely a C interface, a C++ interface and ObjectStore's proprietary *Data Manipulation Language* (DML) interface. It has a variety of features including collections, sets, lists, bags, versioning, workspaces, and performance features like clustering of objects. We are an alpha test site for the ARPA OpenDB, developed by Texas Instruments, and have only recently begun exploring this system, but its open design and modularity seem well suited for our purposes. The *Object System of STONE (OBST)* is a public domain system which provides a C++ interface to the OODBMS. It also has features like collections, sets, lists, bags, and clustering, but lacks facilities for version management in its current release. A special and useful feature it provides is the directory facility that can be used to manage named objects in an UNIX-like directory hierarchy.

The GOOD Common Interface

Based on a review of existing OODBMS capabilties and interface designs, the GOOD was developed with the hopes of spanning several systems. The GOOD common interface to the OODBMS of the IMPORT system was specified using C++ classes. Classes such as *database*, *directory*, *collection*, *set*, *list*, *tree*, *list_cursor*, *set_cursor*, *tree_cursor*, *segment*, *configuration*, and *workspace* provide the necessary database functionality. The main functionality of the various classes is described here; Appendix D contains a more detailed description.

The *database* Class. Databases serve as stores for the objects created by an application. A database is created by sending the message *create* to the database class. Messages such as "open", "close", "look_up" are available for opening, closing, and looking up a database respectively. A database object is specified as a parameter to the persistent version of the C++ *new* operator, the DNEW. The DNEW macro has syntax and semantics similar to that of C++ new operator, in that it takes as an argument a database object and creates the new object in the specified database.

The *directory* Class. The "directory" class facilitates naming of objects, retrieving of named objects, and organization of named objects of a database in a UNIX-like directory hierarchy. A database is provided with a root directory ("/") at database-creation time. Other directories may be created by the application using the "mkdir" message. Messages are available for inserting named objects into directories ("insert"), retrieving named objects ("lookup"), and removing named objects from directories ("rm").

The *list* Class. A list is an aggregate object whose members are maintained in the order in which they were inserted. Messages are provided to facilitate inserting elements in the beginning or at the end of a list ("insert_first" and "insert_last"). Selecting all members of a list that satisfies a certain predicate is possible using the "query" message, which takes as argument a boolean-valued expression that operates on the data fields of the members of the list. Members of the list that return true when the boolean

expression is applied to it are returned by the method. A cursor type called "list_cursor" is available for iterating over lists.

The *set* Class. A set is an aggregate object whose members are not maintained in any particular order. Messages for inserting and removing elements (*insert* and *remove*) are available. A *query* message similar to the one in the *list* class is available for the *set* class also. The *set_cursor* class facilitates iteration over members of a set.

The *tree* Class. The *tree* class provides a mechanism for building multiway rooted trees. Most of the functionality of trees is provided through the tree_cursor class.

The *list_cursor* Class. A list cursor object is used to iterate over members of a list. It is created for a particular list object using the *create* message, which takes a list object as its argument. A list cursor object points to the first element of the list initially. Thereafter, the remaining elements of the list can be visited in order by sending the *next* message to the cursor. Messages *last* and *previous* are also available to traverse the list in the reverse order.

The *set_cursor* Class. A set cursor object is used to iterate over the members of a set. It is created for a particular set using the *create* message, which takes a set object as its argument. It points to some arbitrary element of the set initially. Thereafter, all the elements of the set can be visited by sending the *next* message repeatedly to the cursor until the cursor points to a null object.

The *tree_cursor* Class. A tree cursor object is used to traverse a tree in a manner desired by the user. Like the list cursor and set cursor objects, a tree cursor object is also created for a specific tree object. A tree cursor object points to the root of the tree initially. Thereafter the user can traverse the tree using messages such as *first_child*, *next_sibling*, *prev_sibling*, *first_sibling*, and *parent*. Messages are also available for adding children to a node of the tree (*add_child*) and removing leaves from a tree (*remove*).

The *configuration* Class. A configuration is a grouping of objects that evolve at the same time. In other words, a configuration is the granularity at which versioning can be done in the system. An object is made a member of a configuration when the object is created using the CNEW macro. The CNEW macro creates an object in a configuration as opposed to the DNEW macro, which creates it in a database. A new version of a configuration is created by checking out a configuration to a private workspace using the *checkout* message, and checking it back in using the *checkin* message. This model supports versioning of the latest versions of a configuration, as well as versioning of older versions of a configuration, thus allowing branching version sets possible.

The *workspace* Class. Workspaces allow different groups of users to perform their work without interfering with each other. All configurations initially belong to a global workspace. A global workspace is created using the message *create_global* and child workspaces are created using the message *create*. Configurations are checked out to the child workspaces using the *checkout* message of the configuration class. The *set_current* message sets a workspace as the current workspace. Only configurations checked out to the current workspace may be manipulated by the user.

The *segment* Class. Segments are contiguous chunks of disk space and are used to cluster objects for the purpose of disk storage and retrieval efficiency. Objects that are expected to be used as a group by applications are placed in the same segment when they are created, using the SNEW macro.*

*The SNEW macro takes a segment object as argument and creates a new object in the specified segment.

Current Status and Future Work

The GOOD interface was implemented for the ObjectStore and has been used to implement IMPORT. An implementation for the OpenDB has begun. The OBST system is a candidate as well. Reviews of other OODBMS to support IMPORT continue.

4 FUTURE WORK

Concept Development

IMPORT is being developed concurrently with other applications as Persistent ModSim and ModLog are being fielded. These applications can be readily ported to IMPORT, so that they will be invaluable in testing and in determining needed extensions. As the product enters into testing, newly revealed requirements will give direction to future work.

A number of features found in other languages suggest avenues for research. For example, the class notion might be extended to support *static* fields and methods, as in C++. The introduction of generics is another. An implementation of the *association* concept from OMT has some precedence in research literature (Rumbaugh 1987), and would permit a direct mapping from OMT style design into code. The concept of the *back-pointer* of ObjectStore is a limited implementation of association. *Reflection* capabilities have been shown to be quite powerful especially for distributed actor computational models and for constraint programming (Foote 1989).

Note that the current implementation is a "first-pass" and must be reworked into a more robust design. The SEE classes are the beginnings of a framework for persistent compilation. For example, the current implementation uses the *union* structure to discriminate ast nodes. This is not an object-oriented approach. Methods for each language construct exist in the current id_semantic_controller class and its inverse, the id_code_generator class. A more flexible object-oriented approach is the "Walker" design pattern (Gamma 1993). In this approach, each ast node has a "Traverse'" method that takes a "Walker'" object as an argument. Each ast node would have one method that is sent to a walker passing the node as an argument. The walker is then responsible for acting on the node. This approach keeps the underlying representation simple and collects all the operations on it, in one place. Many other design patterns could be equally beneficially when applied to the classes.

A major piece of work that must be done to support practical applications development is to construct an extensive set of class libraries. The INTERFACE module facility and the availability of large bodies of existing C and C++ code should make this task easy to accomplish.

The current simulation runtime is a sequential model. Also being evaluated are implementation strategies for parallel runtimes, such as Time Warp (Jefferson 1985). Another possible approach is the extension of object database facilities to support efficient roll-back mechanisms using fine-grain versioning (Herring 1991). The general area of *Parallel and Distributed Simulation* (PADS) is also an active field of research that offers many new approaches (Steinman 1992). Future research will include creating extensions to the runtime to support real-time device control, as discussed in *Extensions*, p 51.

Many issues relate to the database-centered implementation of IMPORT. The use of the object repository has opened up many research avenues in configuration management (Kalathil 1993). Also, the Common Object Request Broker Architecture (OMB 1991) combined with the high band width supported by the Defense Simulation Internet offers an approach to support distributed interactive simulation (Martin 1992). IMPORT could be implemented as a language server with programming interfaces (editors) serving as clients in a heterogeneous network compilation environment.

Integrated Simulation Language Environment

The IMPORT/DOME language is one of a number of tools that comprise the Integrated Simulation Language Environment (ISLE). The tools in the ISLE environment all interact with the persistent object repository. The object repository is the heart of the environment, and stores information about the class hierarchy, the programs under development, and the results of program execution; all at the object-level of granularity (as opposed to the file, or function-level granularities that exist in most systems). Figure 20 shows the flow of information between components of ISLE.

The object repository (OR) is currently realized by using a commercial object-oriented database. Already defined is a set of generic access primitives (referred to as Generic Object-Oriented Database, or GOOD) that isolate ISLE from dependence upon any particular database. Currently, the generic interface supports ObjectStore (Objectstore 1991), and ARPA's OpenDB (Wells 1992). The Obst object-oriented database (Stone 1992) is being evaluated as a possible host.

An object framework of classes has been defined that model the intermediate forms of the IMPORT/DOME compilation structures. These classes constitute the basic data structures necessary to support the software engineering environment. They permit storage in the object repository of the IMPORT/DOME programs, and provide a common access mechanism for the entire ISLE toolset.

Storing programs at the object-level of granularity allows new approaches to configuration management and version control by allowing a finer control of the relationships and interdependencies between objects than is available through file-based systems. The ISLE Versionarian maintains a dependency network between object artifacts in the object repository and mediates access for the other ISLE tools. The object repository organizes versions of modules into compatible configurations. The

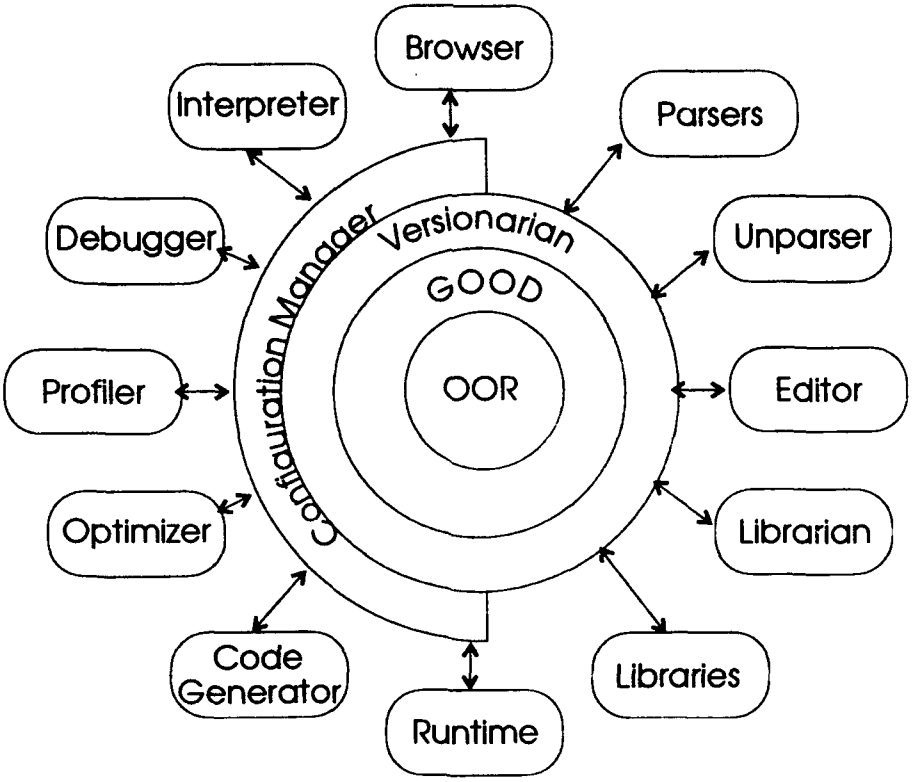

Figure 20. Components of the ISLE Architecture.

Versionarian uses both information about dependencies between software objects and about the differences between versions of software objects to automate a number of configuration management activities, including: determining if a certain configuration is compatible, generating a set of compatible configurations satisfying certain criteria, and determining the impact of proposed changes to a software object in a configuration (Kalathil 1993).

A Compilation Manager (CM) assembles an executable program from the network of objects within a configuration. It interacts with the Versionarian to ensure that a consistent network of source objects are selected. It is also responsible for optimizing the virtual methods to eliminate the overhead associated with function-table lookup when the methods are not overloaded. The Compilation Manager can use the capabilities of the Versionarian that maintain dependency information at object-level granularity to facilitate the implementation of smart recompilation algorithms (Tichy 1986).

Many of the tools, e.g., the Interpreter, Debugger, and Profiler, depend on the Compilation Manager. The interpreter provides an interactive environment for exploratory (or bottom-up) programming. The Debugger helps identify and correct programming problems. The Profiler helps create efficient programs by alerting the user to the most computationally intensive methods. All of these tools interact with the object repository (through the CM and Versionarian) and operate from the intermediate form of IMPORT/DOME programs.

The remaining tools work to convert source input into the intermediate representation, and currently consist of the IMPORT/DOME parser, the unparser, editor, and librarian. The parser takes object specifications as input and, working in conjunction with the Versionarian, places intermediate-form objects into the object repository (Figure 20). The unparser retrieves intermediate-form objects and translates them back into source for the editor. The editor understands the syntax of the language, and supports the direct manipulation and editing of these intermediate-form objects. Finally, the librarian is a knowledge-based programming assistant that helps the user to navigate the class hierarchy and find applicable object definitions.

5 SUMMARY

This work has described IMPORT/DOME, a new language system designed and implemented to provide full integration of previously developed software technologies to support general modeling and simulation. IMPORT is an integrated application of an object-oriented, imperative and declarative programming language that combines process-based discrete-event simulation and persistent object storage to address large-scale, complex systems modeling.

IMPORT is characterized by the following:

- It introduces two new module types: the KNOWLEDGE and the INTERFACE modules.
- Records, procedures, pointers, and subrange types have been removed from IMPORT.
- IMPORT is a strongly typed language, and variables, constants, classes, and methods must be declared before being used.
- Method bodies are the only places where imperative or declarative code can be found.
- IMPORT uses dynamic memory management—built-in procedures for the creation of object instances.
- IMPORT inherits, and augments, the built-in procedures and functions of ModSim.
- At present, IMPORT depends of *Standard Libraries* for simulation support and persistence.
- IMPORT is a fully functional programming language with an implementation based on an object-oriented database.

The current implementation of IMPORT is a "first pass," being further developed as other applications are being fielded. As product is being tested and validated, new avenues for research and product improvement are being defined and explored.

REFERENCES

Ada 9X Requirements Rationale, Ada 9X Project Report (Office of the Under Secretary of Defense for Acquisition, Washington, DC, 1991).

Aho, Alfred V., Ravi Sethi, and Jeffrey D. Ullman, *Compilers, Principles, Techniques, and Tools* (Addison-Wesley, 1986).

Atkinson, M.P., et al., "An Approach to Persistent Programming," *Readings in Object-Oriented Database Systems* (Morgan Kaufmann, Palo Alto, CA, 1989).

Brown, Alan W., *Object-Oriented Databases: Applications in Software Engineering (International Series in Software Engineering)* (McGraw-Hill, 1991).

CACI Products Company, *ModSim II: The Language for Object-Oriented Programming* (CACI Products Company, La Jolla, CA, 1992).

REFERENCES (Cont'd)

Chin Guan Teo, Joseph, *Construction of a Two-Phase Compiler Using an Object-Oriented Database*, Master's Thesis (University of Illinois at Urbana-Champaign, 1992).

Dart, Susan A., *Parallels in Computer-Aided Design Framework and Software Development Efforts*, CMU/SEI-92-TR-9 (Software Engineering Institute, Carnegie-Mellon University, May 1992).

Department of Army, *Army Technology Base Master Plan*, Vol 1 and 2 (February 1992).

Director of Defense Research and Engineering, *DoD Key Technologies Plan* (July 1992).

DOD Simulations: Improved Assessment Procedures Would Increase the Credibility of Results (United States General Accounting Office, Washington, DC, 1987).

Foote, Brian, and Ralph E. Johnson, "Reflective Facilities in Smalltalk-80," *Object Oriented Programming: Systems, Languages, and Applications* (October 1989).

Herring, Charles, Biju Kalathil, and Joseph Teo, *Research in Persistent Simulation: Persistent ModSim*, Draft Technical Report (USACERL, 1993).

Herring, Charles, Jeffrey Wallace, and R. Alan Whitehurst, *Application of Object-Oriented Programming to Combat Modeling and Simulation*, P-91/46/ADA242673 (USACERL, September 1991).

Herring, Charles, and R.A. Whitehurst, "Adding Persistence to An Object-Oriented Simulation Language," *Society for Computer Simulation Multiconference on Object-Oriented Simulation* (Simulation Councils, Inc., San Diego, CA, 1991).

Herring, Charles, and R. Alan Whitehurst, *Letter Report to U.S. Army TRADOC Analysis Command* (USACERL, 1991).

Jefferson, David, *Fast Concurrent Simulation Using the Time Warp Mechanism* (Society for Computer Simulation, San Diego, CA, January 1985).

Kalathil, Biju J., and Charles Herring, *System Support for Assembling Compatible Configurations in an Integrated Programming Environment* (Submitted to Software Configuration Management, May 1993).

Krasner, G.E., and S.T. Pope, "A Cookbook for Using the Model-View-Controller User Interface Paradigm in Smalltalk-80," *Journal of Object-Oriented Programming* (August 1988), pp. 26-49.

Lubars, M., and M.T. Harandi, "Knowledge-Based Software Design Using Design Schemas," *Proceedings of the 9th International Conference on Software Engineering* (March 1987), pp. 253-262.

Moss, J.E.B., "Object-Orientation as Catalyst for Language-Database Integration," *Object-Oriented Concepts, Databases, and Applications* (ACM Press, Reading, MA, 1989).

ObjectStore Technical Overview (Object Design, Inc., 1990).

ORB Task Force, *The Common Object Request Broker: Architecture and Specification*, OMG Document Number 91.12.1 (Object Management Group, December 1991).

Rumbaugh, James, "Relations as Semantic Constructs in an Object-Oriented Language," *Object Oriented Programming: Systems, Languages, and Applications* (October 1987).

Sterling, Leon, and Ehud Shapiro, *The Art of Prolog* (MIT Press, Cambridge, MA, 1986).

Tools.h++: Introduction and Reference Manual, V5.1 (Rogue Wave Software, Inc., Corvallis, OR, 1992).

Wells, David L., Jose A. Blakeley, and Craig W. Thompson, "Architecture of an Open OODB," *IEEE Computer* (October 1992).

Whitehurst, R. Alan, Charles Herring, John Pietrzak and Joseph Teo, "Integrating Object Technology for General Purpose Simulation," *Object Oriented Programming: Systems, Languages, and Applications* (October 1993).

Whitehurst, R. Alan, "Simulation Utilizing an Interpretive Object-Oriented Rule-Based Approach," *Object-Oriented Simulation* (Computer Simulation Society, January 1991).

Zdonik, S., and D. Maier, *Readings in Object-Oriented Databases* (Morgan Kaufmann, San Mateo, CA, 1989).

Appendix A

IMPORT Context Free Grammar

This appendix contains the context free grammar for IMPORT. The syntax is represented in the concise Bachus-Naur Form. No semantics are attached to the symbols. The following regular expression operators are used as a notational convenience.

An asterisk	(*) denotes 0 or more symbols.
A plus	(+) denotes 1 or more symbols.
A question mark	(?) denotes 0 or 1 symbols.

Terminal symbols (reserved words) are shown in all capital letters here. However, the current IMPORT implementation is case insensitive to reserved words. Identifiers cannot be reserved words. The start symbol is **program**.

A.1 Module Definitions and Import Statement

```
program            ⟶ (definition_module
                      | implementation_module
                      | interface_module
                      | key)+

definition_module  ⟶ DEFINITION MODULE identifier semi
                      (import_statement)*
```

```
                        (const_decl | type_decl | obj_type_decl)*
                        END MODULE period

implementation_module⟶ IMPLEMENTATION MODULE identifier semi
                        (import_statement)*
                        (obj_methods_decl)*
                        END MODULE period

interface_module    ⟶ INTERFACE MODULE identifier semi
                        (ext_obj_methods_decl)*
                        END MODULE period

key                 ⟶ KEY identifier colon
                        FROM identifier CREATE identifier
                        INVOKE identifier args period

import_statement    ⟶ FROM identifier IMPORT (import_item_list)+ semi

import_item_list    ⟶ import_item| import_item comma import_item_list

import_item         ⟶ (ALL)? identifier (as_item)?

as_item             ⟶ AS identifier (l_paren enum_item_list r_paren)?

enum_item_list      ⟶ identifier (AS identifier)?
                        | identifier (AS identifier)? comma enum_item_list
```

A.2 Constant Declarations

```
const_decl          ⟶ CONST const_id_list

const_id_list       ⟶ const_id const_id_list
                    |  ϵ

const_id            ⟶ identifier eql expression semi
```

A.3 Type Declaration

type_decl ⟶ TYPE type_id_list

type_id_list ⟶ type_id type_id_list
| obj_type_decl type_id_list
| ϵ

type_id ⟶ identifier eql type semi

type ⟶ identifier
| enum_type
| INTEGER
| REAL
| BOOLEAN
| STRING
| CHAR
| LIST
| ADDRESS
| ARRAY l_bracket integer r_bracket OF type

enum_type ⟶ l_paren enum_type_list r_paren

enum_type_list ⟶ identifier
| identifier comma enum_type_list

A.4 Variable Declaration

var_decl ⟶ VAR var_id_list

var_id_list ⟶ var_id semi var_id_list
| ϵ

var_id ⟶ identifier (comma identifier)* colon type

A.5 Object Definitions

obj_type_decl ⟶ identifier eql OBJECT (super_classes)? semi
obj_type_body
END OBJECT semi
| identifier eql OBJECT semi EXTERNAL semi
ext_obj_type_body
END OBJECT semi
| identifier eql OBJECT semi FORWARD semi

super_classes ⟶ l_paren super_class_list r_paren

super_class_list ⟶ identifier
| identifier comma super_class_list

obj_type_body ⟶ field_method_list
(private_list)?
(override_list)?

field_method_list ⟶ ϵ
| var_id semi field_method_list
| method_prototype semi field_method_list

private_list ⟶ PRIVATE field_method_list

override_list ⟶ OVERRIDE field_method_list

method_prototype ⟶ ask_meth_proto
| tell_meth_proto
| query_meth_proto
| op_meth_proto
| const_meth_proto
| dest_meth_proto

ask_meth_proto ⟶ ASK METHOD identifier
parameters
(return_type)?

tell_meth_proto ⟶ TELL METHOD identifier
parameters

query_meth_proto ⟶ QUERY METHOD identifier
parameters
colon BOOLEAN

op_meth_proto ⟶ OPERATOR METHOD operator
parameters
(return_type)?

const_meth_proto ⟶ CONSTRUCTOR METHOD ObjInit
parameters

dest_meth_proto ⟶ DESTRUCTOR METHOD ObjTerminate l_paren r_paren

parameters ⟶ l_paren (parameter_list)? r_paren

parameter_list ⟶ formal_parameter
| formal_parameter semi parameter_list

formal_parameter ⟶ IN var_id
| OUT var_id
| INOUT var_id

```
return_type      ⟶ colon type

operator         ⟶ times | divide | plus | minus | eql
                 |  neq | lss | gtr | geq | assign
                 |  pluseql | minuseql | timeseql | diveql
```

A.6 Object Declaration

A.6.1 Object Declaration in Implementation Modules

```
obj_methods_decl ⟶ OBJECT identifier semi (method_decl)*
                   END OBJECT semi

method_decl      ⟶ method_prototype semi (var_decl | const_decl)*
                   statement_block METHOD semi
```

A.6.2 Object Declaration in Interface Modules

```
ext_obj_methods_decl ⟶ OBJECT identifier (AS identifier)? (IN language)? semi
                       (ext_field_rename | ext_meth_proto)*
                       END OBJECT semi

language         ⟶ C++
                 |  C

ext_obj_type_body ⟶ var_id semi ext_field_method_list
                  |  ext_def_meth_proto semi ext_field_method_list——
ext_field_rename  ⟶ identifier (AS identifier)? semi
```

ext_def_meth_proto ⟶ ask_meth_proto
| op_meth_proto
| const_meth_proto
| dest_meth_proto

ext_meth_proto ⟶ ext_ask_meth_proto
| ext_op_meth_proto
| ext_const_meth_proto
| ext_dest_meth_proto

ext_ask_meth_proto ⟶ ASK METHOD identifier
(AS identifier)? semi (STATIC semi)?

ext_op_meth_proto ⟶ OPERATOR METHOD operator
(AS operator)? semi (STATIC semi)?

ext_const_meth_proto ⟶ CONSTRUCTOR METHOD ObjInit
(AS identifier)? semi (STATIC semi)?

ext_dest_meth_proto ⟶ DESTRUCTOR METHOD ObjTerminate
(AS identifier)? semi (STATIC semi)?

A.7 Expressions

assignment ⟶ location assign expression
| location pluseql expression
| location minuseql expression
| location timeseql expression
| location diveql expression

expression ⟶ integer
| real
| string

| char
| list
| location
| NULL
| UNINSTANTIATED

| minus expression %prec uminus
| expression minus expression
| expression plus expression
| expression times expression
| expression divide expression
| expression DIV expression
| expression MOD expression
| l_paren expression r_paren

| boolval
| NOT expression
| expression AND expression
| expression OR expression

| expression eql expression
| expression neq expression
| expression lss expression
| expression leq expression
| expression gtr expression
| expression geq expression

| ask_method_invocation
| tell_method_invocation
| operator_method_invocation
| query_method_invocation
| built_in_func

location ⟶ (inherited_inherited_from)? identifier
| SELF
| location l_bracket expression r_bracket

```
boolval          ⟶ TRUE
                 |  FALSE

list             ⟶ l_bracket expression (comma expression)* r_bracket
```

A.8 Method Invocations

```
inherited_from          ⟶ FROM identifier

inherited_inherited_from ⟶ INHERITED FROM identifier

method_invocation       ⟶ ask_method_invocation
                        |  tell_method_invocation
                        |  query_method_invocation

ask_method_invocation   ⟶ ASK expression (TO)? identifier
                           args (inherited_from)?

tell_method_invocation  ⟶ TELL expression (TO)? identifier
                           args (inherited_from)?

query_method_invocation ⟶ QUERY identifier (ABOUT)? identifier
                           args (inherited_from)?

args                    ⟶ l_paren (arg_list)? r_paren

arg_list                ⟶ (expression comma)* expression
```

A.9 Built-In Functions

```
built_in_func  ⟶ abs| cap| chartostr| chr| float| inttostr| lower
               |  max| maxof| min| minof| odd| ord| persistent| position
               |  realtostr| round| schar| strcat| strlen| strput| strtoint
               |  strtoreal| substr| trunc| upper| val
               |  simtime

abs            ⟶ ABS l_paren expression r_paren
cap            ⟶ CAP l_paren expression r_paren
chartostr      ⟶ CHARTOSTR l_paren expression r_paren
chr            ⟶ CHR l_paren expression r_paren
float          ⟶ FLOAT l_paren expression r_paren
inttostr       ⟶ INTTOSTR l_paren expression r_paren
lower          ⟶ LOWER l_paren expression r_paren

max            ⟶ MAX l_paren scalar_type r_paren
maxof          ⟶ MAXOF l_paren arg_list r_paren
min            ⟶ MIN l_paren scalar_type r_paren
minof          ⟶ MINOF l_paren arg_list r_paren

odd            ⟶ ODD l_paren expression r_paren
ord            ⟶ ORD l_paren expression r_paren
persistent     ⟶ PERSISTENT l_paren expression r_paren
position       ⟶ POSITION l_paren expression comma expression r_paren
realtostr      ⟶ REALTOSTR l_paren expression r_paren
round          ⟶ ROUND l_paren expression r_paren
schar          ⟶ SCHAR l_paren expression comma expression r_paren
simtime        ⟶ SIMTIME l_paren r_paren
strcat         ⟶ STRCAT l_paren arg_list r_paren
strlen         ⟶ STRLEN l_paren expression r_paren
strput         ⟶ STRPUT l_paren arg_list r_paren
strtoint       ⟶ STRTOINT l_paren expression r_paren
strtoreal      ⟶ STRTOREAL l_paren expression r_paren
substr         ⟶ SUBSTR l_paren expression comma expression comma
                 expression r_paren
trunc          ⟶ TRUNC l_paren expression r_paren
```

```
upper          ⟶ UPPER l_paren expression r_paren
val            ⟶ VAL l_paren scalar_type comma expression r_paren
               |  VAL l_paren identifier comma expression r_paren

scalar_type    ⟶ INTEGER
               |  REAL
               |  CHAR
               |  BOOLEAN
```

A.10 Built-in Procedures

```
built_in_proc  ⟶ abort | dec| halt| dispose| inc| input| insert
               |  new| output| replace| strtochar
               |  interrupt| interrupt_all

abort          ⟶ ABORT
dec            ⟶ DEC l_paren expression r_paren
               |  DEC l_paren expression comma expression r_paren
dispose        ⟶ DISPOSE l_paren expression r_paren
halt           ⟶ HALT
inc            ⟶ INC l_paren expression r_paren
               |  INC l_paren expression comma expression r_paren
input          ⟶ INPUT l_paren (arg_list)? r_paren
insert         ⟶ INSERT l_paren expression comma expression comma
                  expression r_paren
new            ⟶ NEW l_paren location (comma expression)? r_paren args
output         ⟶ OUTPUT l_paren (arg_list)? r_paren
replace        ⟶ REPLACE l_paren expression comma expression comma
                  expression comma expression r_paren
strtochar      ⟶ STRTOCHAR l_paren expression comma expression r_paren
interrupt      ⟶ INTERRUPT l_paren expression comma expression r_paren
interrupt_all  ⟶ INTERRUPTALL l_paren expression coma expression r_paren
               |  INTERRUPTALL l_paren expression r_paren
```

A.11 Statements

```
statement          ⟶ assignment
                   |  method_invocation
                   |  built_in_proc
                   |  block_statement
                   |  if_statement
                   |  case_statement
                   |  while_statement
                   |  repeat_statement
                   |  for_statement
                   |  loop_statement
                   |  exit_statement
                   |  return_statement
                   |  terminate_statement
                   |  wait_statement
                   |  transaction_statement
                   |  abort_statement

statement_list     ⟶ statement semi statement_list
                   |  ϵ

block_statement    ⟶ BLOCK (const_decl| type_decl)* statement_block BLOCK

statement_block    ⟶ BEGIN statement_list END semi

if_statement       ⟶ IF expression statement_list
                      (elsif_list)?
                      (else_clause)?
                      END IF

elsif_list         ⟶ elsif_clause
                   |  elsif_clause elsif_list
```

elsif_clause ⟶ ELSIF expression statement_list

else_clause ⟶ ELSE statement_list

case_statement ⟶ CASE expression when_list (otherwise_clause)? END CASE

when_list ⟶ when_clause
| when_clause when_list

when_clause ⟶ WHEN case_label colon statement_list

otherwise_clause ⟶ OTHERWISE statement_list

case_label ⟶ integer
| boolval
| char

while_statement ⟶ WHILE expression statement_list END WHILE

repeat_statement ⟶ REPEAT statement_list UNTIL expression

for_statement ⟶ FOR location assign expression TO expression
(by_clause)? statement_list END FOR
| FOR location assign expression DOWNTO expression
(by_clause)? statement_list END FOR

by_clause ⟶ BY expression

loop_statement ⟶ LOOP statement_list END LOOP

exit_statement ⟶ EXIT

return_statement ⟶ RETURN (expression)?

terminate_statement ⟶ TERMINATE

wait_statement ⟶ WAIT DURATION expression statement_list
(interrupt_clause)? END WAIT
| WAIT FOR expression TO identifier args (inherited_from)?
statement_list (interrupt_clause)? END WAIT
| WAIT FOR TRIGGER expression

interrupt_clause ⟶ ON INTERRUPT statement_list

transaction_statement ⟶ TRANSACTION statement_list (abort_clause)?
END TRANSACTION

abort_clause ⟶ ON ABORT statement_list

abort_statement ⟶ ABORT
| ABORT ALL

A.12 Query Statement

query_statement ⟶ query_goal period
| (IF)? query_goallist
THEN query_goal period

query_goallist ⟶ query_goallist query_connector query_goal
| query_goal

query_goal ⟶ identifier .
| identifier l_paren query_goallist r_paren
| identifier assign query_simplexpr
| integer
| real
| char
| string
| l_bracket r_bracket

```
                    |   query_conslist

query_conslist      ⟶  l_bracket query_consexprlist r_bracket
                    |   l_bracket query_consexprlist '—' identifier r_bracket

query_consexprlist  ⟶  query_consexprlist comma query_consexpr
                    |   query_consexpr

query_consexpr      ⟶  query_goal

query_simexprlist   ⟶  query_simexprlist query_connector query_simplexpr
                    |   query_simplexpr

query_simplexpr     ⟶  query_term query_addop query_term
                    |   query_term

query_addop         ⟶  plus
                    |   minus

query_term          ⟶  query_factor query_mulop query_factor
                    |   query_factor

query_mulop         ⟶  times
                    |   divide
                    |   ^
                    |   '
query_factor        ⟶  integer
                    |   real
                    |   identifier
                    |   l_paren query_simplexpr r_paren
                    |   identifier l_paren query_simexprlist r_paren

query_connector     ⟶  AND
                    |   comma
```

A.13 Comments

comment ⟶ nested_comment
| single_line_comment

nested_comment ⟶ (l_brace nested_comment r_brace)+

single_line_comment ⟶ minus minus (any_character)*

A.14 Operator Symbols

times ⟶ *
divide ⟶ /
plus ⟶ +
minus ⟶ -
eql ⟶ =
neq ⟶ <>
lss ⟶ <
leq ⟶ <=
gtr ⟶ >
geq ⟶ >=
assign ⟶ :=
pluseql ⟶ + =
minuseql ⟶ - =
timeseql ⟶ * =
diveql ⟶ / =

A.15 Punctuation

All symbols on the right hand side are terminals.

l_paren ⟶ (
r_paren ⟶)
semi ⟶ ;
colon ⟶ :

comma → ,
l_bracket → [
r_bracket →]
l_brace → {
r_brace → }
period → .

Appendix B

IMPORT Database Class Library

Support for persistent object storage in IMPORT is provided through a library of pseudo classes. These classes were first prototyped in the PERSISTENT MODSIM development effort [13]. They were further refined in the Generic Object Oriented Database (GOOD) interface developed for implementing IMPORT described in Appendix ??. These IMPORT pseudo classes are implemented through use of the GOOD interface. Hence, persistence in the IMPORT language is supported through the same underlying mechanizm as is its implementation. Porting the GOOD to other database hosts ports the langauge manipulation facilities (parser, code generator, etc.) as well as the language's ability to manipulate persistent storage.

B.1 Overview

It became apparent in the development of PERSISTENT MODSIM that it is necessary for programmers to know and understand the consequences of object persistence. This capability dramatically changes the manner in which programs are designed and implemented. Therefore it was decided that more access to the underlying database functionality was required and that it be presented to programmers in a consistent manner. We chose to provide the interface to persistent storage in PERSISTENT MODSIM through use of a database class library. This same approach is taken in IMPORT.

Through the database class library the essential features of database functionality are presented as objects to the user. This approach is widely

used and permits control over how much detail of the underlying database model is revealed. The class library represents a simplified model of object database. This is consistent with the object paradigm of adding extended features through classes that can be inherited from. It permits programmers the flexibility to develop further specializations based on this model for specific purposes. A consequence of this approach is the necessity to introduce two new concepts into the language in the form of extended syntax. These two extensions are concerned with the specification of persistent allocation of objects (the overloaded NEW) and transaction management (the TRANSACTION statment) to permit concurrent access to databases by multiple users.

This appendix provides documentation of the database model of IMPORT. The model consists of a set of classes representing database functionality to the programmer. The database class library consists of the following classes: Database, Directory, Segment, Collection, Cursor, Configuration, and Workspace. Each class is implemented as in separate DEFINITION file which can be imported into any IMPORT program. These classes have no restrictions on them, they can be inherited from and their methods overridden as necessary. However, there are no corresponding IMPLEMENTATION files. The code implementing these classes is contained in the GOOD.

B.2 Databases

IMPORT variables of the object type (class) Database represent databases. Databases can be thought of as files that store objects. In an IMPORT program, an instance of a Database object is associated with one physical database at any given time. Database objects have no fields–only methods. These methods implement the basis notions associated with database operations. The methods of database objects include create, open, close, and delete. There are methods for determining the status of the database as well as methods associated with the functioning of the other classes in the database library.

The class database allows programs to create and manipulate persistent objects. Instances of this class are used a parameters in calls to NEW to specify were new persistent objects will be allocated. An open count is maintained for each database and represents the number of times it's Open method was called during the current process. When the open count is set to 0 the database is closed. All databases are automatically closed when the

program terminates. Object instances of this class need not be persistent themselves. The definition module for the Database class is show below. Following is a description of the methods of this class.

```
DEFINITION MODULE Database;

FROM DatabaseRoot IMPORT DatabaseRoot;
FROM Segment IMPORT Segment;

TYPE
    Database =
    OBJECT
        ASK METHOD AllowExternalPointers();
        ASK METHOD RoorDir(): Directory;
        ASK METHOD Create(IN pathname : ARRAY OF CHAR;
                          IN mode : INTEGER;
                          IN if_exists_overwrite : BOOLEAN);
        ASK METHOD Destroy();
        ASK METHOD Open(IN readOnly : BOOLEAN);
        ASK METHOD open(IN pathname : ARRAY OF CHAR;
                          IN readOnly : INTEGER) : Database;
        ASK METHOD Close();
        ASK METHOD GetPathName(OUT pathName : ARRAY OF CHAR);
        ASK METHOD Lookup(IN pathname : ARRAY OF CHAR;
                          IN createMode : INTEGER) : Database;
        ASK METHOD Of(IN item : REFERENCE) : Database;
        ASK METHOD IsOpen(): BOOLEAN;
        ASK METHOD IsWritable(): BOOLEAN;
        ASK METHOD IsEqualTo(IN db1 : Database;
                          IN db2 : Database) : BOOLEAN;
        ASK METHOD CreateSegment() : Segment;
    END OBJECT;

END MODULE
```

AllowExternalPointers: Allow database pointers to cross databases boundaries. After perfoming this operation on a database, the current process and subsequent processes can store cross-database pointers there. When you access a cross-database pointer, if the database it points to is not open, it will be opened automatically.

Close: Decrements the open count of the database. If the open count is 0 the database is closed. If the open count is greater than 0 the database access (read or read/write) is returned to the previous access mode. If this method is called from within a transaction, the open count is not decremented until the end of the current outermost transaction.

Create: Creates a new database with the specified pathname and mode. The values for mode are the same as used in the Unix chmod command. If the parameter if_exists_overwrite is set to true a new database will be created even if one by that name already exists, otherwise a runtime error will occur. A root directory of this database is automatically created if this operation succeeds.

CreateSegment: Creates a segment in the database and returns an object of type Segment.

Destroy: Deletes the database. This method must be called within a transaction.

GetPathName: Returns the pathname of the database. This pathname will always begin with a "/".

IsEaualTo: Returns true if db1 and db2 are the same database.

IsOpen: Returns true if the database is open, otherwise returns false.

IsWritable: Returns true if the database is writable by the current process. Returns false if the database has been opened for read-only or if the process does not have write permission.

LookUp: Associates the database specified in pathname with an instance of the class Database. If the database is not found a runtime error will occur unless createMode is non-zero. If createMode is non-zero and the database was not found a new database will be created. Note this method does not open the database.

Of: This method takes a variable of type object and returns the database object it is stored in.

Open: Opens the database associated with the object. If readOnly is non-zero the database is opened for read access only, otherwise it is open for read/write.

RootDir: A pointer points to the root directory of this database is returned which can be used for subsequent directory maninulation.

B.3 Directories

The Directory class provides for structuring entry points into databases. The named objects in a database are organized in an Unix-like directory hierarchy. A directory may contain other directories and named objects. The default working directory is the root directory ("/") which is created automatically whenever a new database is created. From there, other named objects and directory objects can be manipulated through directory class methods. The methods of directory objects include Mkdir, Rmdir, Getdir, Lookup, Insert, Remove and Ls. Path names to named objects or directory objects are specified either relative to the current working directory or by using the complete path name starting with the root ("/").

```
DEFINITION MODULE Directory;

TYPE
    Directory =
    OBJECT
        ASK METHOD Lookup (IN db : Database;
                           IN pathname : ARRAY OF CHAR) : REFERENCE;
        ASK METHOD Insert (IN dirname : ARRAY OF CHAR;
                           IN item : REFERENCE;
                           IN name : ARRAY OF CHAR);
        ASK METHOD Lookup (IN objname : ARRAY OF CHAR) : REFERENCE;
        ASK METHOD Insert (IN item : REFERENCE;
                           IN objname : ARRAY OF CHAR) : BOOLEAN;
        ASK METHOD Remove (IN objname : ARRAY OF CHAR) : BOOLEAN;
        ASK METHOD Mkdir  (IN dirname : ARRAY OF CHAR) : Directory;
        ASK METHOD GetDir (IN db : Database;
                           IN dirname : ARRAY OF CHAR) : Directory;
        ASK METHOD Rmdir  (IN dirname : ARRAY OF CHAR) : BOOLEAN;
        ASK METHOD Ls ();
     END OBJECT;

END MODULE.
```

Getdir: Gets a directory object with its entry name "dir_name" in the specified database. Returns a pointer to the directory if found, else returns a NULL pointer.

Insert: Inserts the object pointed to by "item" into the directory "dir", with the name "name".

Lookup: Looks up a named object, which can be either an ordinary object or a directory object, in the database. A pointer to the object is returned.

Ls: Returns a list containing the names of the named objects and subdirectories in this directory.

Mkdir: Makes a new subdirectory "dir_name" in this directory, and returns a pointer to this new subdirectory.

Remove: Removes an entry for a named object "obj_name" from this directory. Does not delete the object from the database. Returns 1 if the object has been removed successfully, else 0.

Rmdir: Removes an entry for a named subdirectory "dir_name" from this directory. Does not delete the subdirectory from the database. Returns 1 if the subdirectory has been removed successfully, else 0.

B.4 Collections and Cursors

The Collection module provides classes for grouping objects together and accessing them. The objects contained in these collections are refered to as elements. The following classes are found in the Collection module: List, ParseList, Set, and Tree. Instances of these classes may be allocated in a Database, a Segment, or a Configuration. They have methods to insert and remove objects and for comparing various properties such as equal to and greater than.

The *cursor* classes provide for iteration over the elements of the collection classes. There is a cursor class associated with each of the collection classes. An instance of a cursor object is created for a given instance of a collection object. The cursor object has methods to return specific members of the given collection object. Some of the cursor object's methods are first, last, next, etc. The cursor object also has methods to insert objects into its collection. Any number of cursor objects can be instanciated for a given collection object. Cursor objects are most often allocated transiently.

```
DEFINITION MODULE Collection;
FROM Database IMPORT Database;

TYPE
```

```
List =
OBJECT
    ASK METHOD CreateInDatabase(IN db : Database);
    ASK METHOD IsEmpty(): BOOLEAN;
    ASK METHOD Insert(IN elem : REFERENCE);
    ASK METHOD InsertFirst(IN elem : REFERENCE);
    ASK METHOD Remove(IN elem : REFERENCE);
    ASK METHOD RemoveFirst();
    ASK METHOD RemoveLast();
    ASK METHOD Size() : INTEGER;
END OBJECT;

ListCursor =
OBJECT
    ASK METHOD Create(IN list: List);
    ASK METHOD First(): REFERENCE;
    ASK METHOD InsertAfter(IN ref: REFERENCE);
    ASK METHOD InsertBefore(IN ref: REFERENCE);
    ASK METHOD IsNull(): BOOLEAN;
    ASK METHOD Last(): REFERENCE;
    ASK METHOD More(): BOOLEAN;
    ASK METHOD Next(): REFERENCE;
    ASK METHOD Previous(): REFERENCE;
    ASK METHOD RemoveAt();
    ASK METHOD Retrieve(): REFERENCE;
END OBJECT;

ParseList =
OBJECT
    ASK METHOD Append (IN list : ParseList);
    ASK METHOD CreateInDatabase(IN db : Database);
    ASK METHOD IsEmpty(): BOOLEAN;
    ASK METHOD Insert(IN elem : REFERENCE);
    ASK METHOD InsertFirst(IN elem : REFERENCE);
    ASK METHOD Remove(IN elem : REFERENCE);
    ASK METHOD RemoveFirst();
    ASK METHOD RemoveLast();
 END OBJECT;
```

```
ParseListCursor =
OBJECT
    ASK METHOD Create(IN parseList: ParseList);
    ASK METHOD First(): REFERENCE;
    ASK METHOD InsertAfter(IN ref: REFERENCE);
    ASK METHOD IsNull(): BOOLEAN;
    ASK METHOD More(): BOOLEAN;
    ASK METHOD Next(): REFERENCE;
    ASK METHOD RemoveAt();
    ASK METHOD Retrieve(): REFERENCE;
    ASK METHOD Tail(): ParseList;
END OBJECT;

Set =
OBJECT
    ASK METHOD CreateInDatabase(IN db : Database);
    ASK METHOD IsEmpty(): BOOLEAN;
    ASK METHOD Insert(IN elem : REFERENCE);
    ASK METHOD Remove(IN elem : REFERENCE);
    ASK METHOD Size() : INTEGER;
END OBJECT;

SetCursor =
OBJECT
    ASK METHOD Create(IN set: Set);
    ASK METHOD First(): REFERENCE;
    ASK METHOD IsNull(): BOOLEAN;
    ASK METHOD More(): BOOLEAN;
    ASK METHOD Next(): REFERENCE;
    ASK METHOD RemoveAt();
    ASK METHOD Retrieve(): REFERENCE;
END OBJECT;

Tree =
OBJECT
    ASK METHOD CreateInDatabase(IN db : Database);
    ASK METHOD IsEmpty(): BOOLEAN;
END OBJECT;
```

```
    TreeCursor =
    OBJECT
        ASK METHOD AddChild(IN item : REFERENCE);
        ASK METHOD Create(IN tree: Tree);
        ASK METHOD FirstChild(): REFERENCE;
        ASK METHOD IsFirstSibling(): BOOLEAN;
        ASK METHOD IsLastSibling(): BOOLEAN;
        ASK METHOD IsLeaf(): BOOLEAN;
        ASK METHOD IsRoot(): BOOLEAN;
        ASK METHOD More(): BOOLEAN;
        ASK METHOD NextSibling(): REFERENCE;
        ASK METHOD Parent(): REFERENCE;
        ASK METHOD PrevSibling(): REFERENCE;
        ASK METHOD RemoveAt();
        ASK METHOD Retrieve(): REFERENCE;
        ASK METHOD Root(): REFERENCE;
    END OBJECT;

END MODULE.
```

List

CreateInDatabase: Creates a list in the specified database.

First: Sets the cursor at the first element of the list and returns a pointerto the element.

Insert: Inserts the passed in item at the end of the list.

InsertFirst: Inserts the passed in item at the front of the list.

Remove: Removes the specified item from the list.

RemoveFirst: Removes the first element from the list.

RemoveLast: Removes the last element from the list.

Size: Returns the number of elements in the list.

ListCursor

Create: Creates a list cursor for the specified list.

InsertAfter: Inserts the passed in object after the current element in the list.

InsertBefore: Inserts the passed in object before the current element in the list.

IsNull: Returns 0 (false) if the cursor is located at a valid element of the list; returns 1(true) otherwise.

Last: Positions the cursor at the last elcment of the list and returns a pointer to the element.

More: Returns 1(true) if the cursor is not at the last element of the list; returns 0(false) otherwise.

Next: Advances the cursor to the next element of the list and returns a pointer to the element.

Previous: Moves the cursor to the previous element of the list and returns a pointer to the element.

RemoveAt: Removes from the list the element at which the cursor is currently located.

Retrieve: Returns a pointer to the element of the list at which the cursor is currently located.

ParseList

Append: Appends the passed in list to the end of the target list. This is done by physically attaching the passed in list to the end of the target list and not by making a copy of the passed in list. Hence, changes made to one list may affect the other.

Create: Creates a list in the specified database and returns a pointer to the created list.

Insert: Inserts the passed in item at the end of the list.

InsertFirst: Inserts the passed in item at the front of the list.

Remove: Removes the specified item from the list.

RemoveFirst: Removes the first element from the list.

RemoveLast: Removes the last element from the list.

ParseListCursor

Create: Creates a parse list cursor for the specified list.

First: Sets the cursor at the first element of the list and returns a pointer to the element.

InsertAfter: Inserts the passed in object after the current element in the list.

IsNull: Returns 0(false) if the cursor is located at a valid element of the list; returns 1(true) otherwise.

More: Returns 1(true) if the cursor is not at the last element of the list; returns 0(false) otherwise.

Next: Advances the cursor to the next element of the list and returns a pointer to the element.

RemoveAt: Removes from the list the elemer' at which the cursor is currently located.

Retrieve: Returns a pointer to the element of the list at which the cursor is currently located.

Tail: Returns the tail of the list starting from the current location of the cursor.

Set

Create: Creates a set in the specified database and returns a pointer to the created set.

IsEmpty: Returns 1(true) if the set is empty 0(false) otherwise.

Insert: Inserts the passed in item into the set.

Remove: Removes the specified item from the set.

Size: Returns the number of elements in the set.

SetCursor

Create: A set cursor is created for a particular set which is specified when the set cursor is created.

First: Sets the cursor at the first element of the set and returns a pointer to the element.

IsNull: Returns 0(false) if the cursor is located at a valid element of the set; returns 1(true) otherwise.

More: *Returns 1(true) if the cursor is not* at the last element of the set; returns 0(false) otherwise.

Next: Advances the cursor to the next element of the set and returns a pointer to the element.

RemoveAt: Removes from the set the element at which the cursor is currently located.

Retrieve: Returns a pointer to the element of the set at which the cursor is currently located.

Tree

Create: Creates a tree in the specified database, with the object pointed to by the argument root as the root of the tree. Returns a pointer to the created tree.

IsEmpty: Returns 1(true) if the tree is empty; returns 0(false) otherwise.

TreeCursor

AddChild: Adds the passed in item as a child of the current node.

FirstChild: Advances the cursor to the first child of the current node and returns a pointer to the object stored therein.

IsFirstSibling: Returns true (1) if the current node is the first sibling, false (0) otherwise.

IsLastSibling: Returns true (1) if the current node is the last sibling, false (0) otherwise.

IsLeaf: Returns true (1) if the current node is a leaf, false (0) otherwise.

IsRoot: Returns true (1) if the current node is the root of the tree, false (0) otherwise.

NextSibling: the cursor to the next sibling of the current node, if any, and returns a pointer to the object stored therein.

Parent: Moves the cursor to the parent of the current node and returns a pointer to the object stored therein.

PrevSibling: Advances the cursor to the previous sibling of the current node, if any, and returns a pointer to the object stored therein.

RemoveAt: Removes the current node from the tree, if it does not children. Does nothing otherwise.

Retrive: Returns a pointer to the object stored in the current node of the tree.

Root: Sets the cursor at the root of the tree and returns a pointer to the element.

B.5 Configurations

The Configuration is the unit of version control. Instances of the Configuration class provide a means to specify groupings of objects that are to be treated as a unit for version control. Object instances of any type may be allocated into a given Configuration. This includes objects of type Configuration, thus the programmer can organize subgroups of related objects with configurations to any level. Configuration objects are the unit of both version control and database locking. They can be thought of as long duration transactions on the database. There can be no conflict when a Configuration is checked out to a given application as other applications can also check out versions of the Configuration. Configurations provide the mechanism to achieve change management in shared environments. Some of the methods of the class Configuration include check out, check out on a branch, and check in. As versions of a configuration are checked out, changed, and checked back in, a version tree is formed that permits users to go back to any previous version. There are methods for traversing the version tree of a particular Configuration object to retrieve past versions. In all of the methods of the Configuration class if the parameter recursive is true, the function of that method will be applied to all subconfigurations of the given configuration. Note that Configurations must be used within the context of Workspaces.

```
DEFINITION MODULE Configuration;

FROM Segment IMPORT Segment;
FROM Workspace IMPORT Workspace;
FROM Database IMPORT Database;

TYPE
    Configuration =
    OBJECT
        ASK METHOD Create (IN db: Database;
                           IN name : ARRAY OF CHAR) : Configuration;
        ASK METHOD Checkin (IN db : Database;
                            IN name : ARRAY OF CHAR;
                            IN recursive : BOOLEAN);
        ASK METHOD Checkout (IN db : Database;
                             IN name : ARRAY OF CHAR;
                             IN recursive : BOOLEAN);
        ASK METHOD CheckoutBranch (IN db : Database;
                                   IN name : ARRAY OF CHAR;
                                   IN branchName : ARRAY OF CHAR;
                                   IN versionName :ARRAY OF CHAR;
                                   IN recursive : BOOLEAN);
        ASK METHOD Resolve (IN item : REFERENCE;
                            IN name : ARRAY OF CHAR): REFERENCE;
        ASK METHOD Lookup (IN db : Database;
                           IN name : ARRAY OF CHAR) : Configuration;
        ASK METHOD Merge (IN db : Database;
                          IN ws : Workspace;
                          IN name : ARRAY OF CHAR);
        ASK METHOD Successor (IN db : Database;
                              IN name : ARRAY OF CHAR) : Configuration;
        ASK METHOD Predecessor (IN db : Database;
                                IN name : ARRAY OF CHAR) : Configuration;
    END OBJECT;

END MODULE.
```

Checkin: Removes the current version of the configuration from the current workspace, puts it in the parent workspace, freezes it, and makes it current for the parent on the branch that contains it.

Checkout: Creates a new version of the configuration and inserts it into the current workspace. If the parameter recursive is true, all subconfigurations of the configuration are checkout as well.

CheckoutBranch: Creates a new version of the configuration and inserts it into the current workspace, but a new branch will be created.

Create: Creates a configuration in the specified database. This creates a branch containing the newly created configuration. This initial version of the configuration is set to the default for the current workspace.

Merge: Checks out the specified configuration and calls SetSuccessor to set it to the new version.

Predecessor: Returns a pointer to the configuration that is the predecessor of the current configuration. This does not change the current configuration in the workspace. This method returns false if there is no predecessor.

Resolve: Takes a pointer to an object in one version of a configuration and returns a pointer to the corresponding version of the same object in another version of the configuration.

Successor: Returns a pointer to the successor of the specified configuration.

B.6 Workspaces

The Workspace class is related to the use of Configurations. Workspace objects provide a way to structure shared and private access to Configuration objects. All manipulation of Configurations must take place within a current workspace. Configurations are checked in and out of workspaces. Calls to the various check out methods of Configuration objects are relative to the current workspace. Workspace objects are linked (or nested) hierarchically into a workspace tree. Applications can set the access privileges to parts of this workspace tree to control access (and hence change). There is must be a "global workspace". Workspace objects are then allocated within the context of this global workspace. Workspaces combined with Configurations supply the needed concepts for computer supported collaborative work. The Workspace class include methods to create child workspaces of a parent workspace, to get the parent of given workspace, and to set a Workspace

object to be the current workspace.

```
DEFINITION MODULE Workspace;

FROM Database IMPORT Database;

TYPE
    Workspace =
    OBJECT
        ASK METHOD CreateGlobal (IN db : Database;
                                 IN name : ARRAY OF CHAR) : Workspace;
        ASK METHOD Create (IN db : Database;
                           IN name : ARRAY OF CHAR;
                           IN parent : Workspace) : Workspace;
        ASK METHOD Resolve (IN db : Database;
                            IN name : ARRAY OF CHAR;
                            IN item : REFERENCE) : REFERENCE;
        ASK METHOD Lookup (IN db : Database;
                           IN pathname : ARRAY OF CHAR) : Workspace;
        ASK METHOD SetCurrent (IN ws : Workspace);
        ASK METHOD Current () : Workspace;
        ASK METHOD GetParent () : Workspace;
        ASK METHOD GetName (OUT name : ARRAY OF CHAR);
        ASK METHOD Of (IN item : REFERENCE) : Workspace;
        ASK METHOD Resolve (IN item : REFERENCE): REFERENCE;
    END OBJECT;

END MODULE.
```

Create: Creates a child of the current workspace in the specified database. This new child workspace can be referenced by the specified name.

CreateGlobal: Creates a global workspace in the specified database and with the specified name. This is the root workspace from which all child workspaces are rooted.

Current: Returns a pointer to the current workspace object.

GetName: Returns the name of the current workspace.

GetParent: Returns a pointer to the parent workspace of the current workspace. This method does not make the parent current.

Lookup: Returns a pointer to the serched workspace in the database.

Of: Returns a pointer to a workspace in which the specified object resides.

Resolve: Returns a pointer to the version of the specified object made visible by the current workspace.

SetCurrent: Sets the workspace to be the current workspace. Note, this takes effect at the beginning of the next transaction.

B.7 Segments

Databases are composed of some number of Segments. Segments can be thought of as the smallest unit of memory that is transferred from persistent to transient storage. Every database is created with an initial segment. As objects are stored in the database additional segments are created automatically. The Segment class is provided as a means of clustering groups of objects for performance reasons. As a segment is the unit of memory transfer, significant performance improvements can be gained by physically collocating related objects. One of the methods of the Database class is the creation of Segments. This method returns a Segment object. Methods of the Segment class include those to determine state information, control size, and destruction.

```
DEFINITION MODULE Segment;

TYPE
   Segment =
   OBJECT
        ASK METHOD Create(IN db : Database) : Segment;
        ASK METHOD DatabaseOf() : Database;
        ASK METHOD Destroy();
        ASK METHOD Of(IN item : REFERENCE) : Segment;
   END OBJECT;

END MODULE.
```

Create: Returns a pointer to the newly created segment in the specified database.

DatabaseOf: Returns a pointer to the Database object containing this segment.

Destroy: Deletes this segment from the database.

Of: Returns a pointer to the segment object containing the specified object.

Appendix C

Software Engineering Classes

This appendix provides a listing of C++ header files describing the classes that make up a persistent object framework used to model the IMPORT language. This model is called an *intermediate representation.* We refer to these classes as Software Engineering Environment Classes because they provide the basis for implementation of the language and its support tools. These classe are used in conjunction with Yacc++ [29] and the GOOD interface (see Appendix ??) to implement an object repository for IMPORT programs. Yacc++ is used to build the lexer and parser. The classes, through the GOOD interface, provide for persistent storage and manipulation of IMPORT compilation artifacts.

The following header files are displayed below: id_seec.h, id_type_expr.h, id_symtab.h and i_semantics.h. The id_seec.h file defines the IMPORT/DOME software engineering environment support classes. These classes provide a model of the basic features of the language: the module, class, method and the declaration. Also contained in this file are the abstract syntax node (ast) and ast list classes. The second section consists of the id_type_expr.h file. This file contains the type_expression class and its associated list class. The symbol table (Symtab) and support classes, the scope, symtab_entry and symtab_entry_list are shown in the next section. Finally, the semantic action controller class, id_semantic_controller, is shown. This class implements the semantics of the language and calls to methods of this class are triggered by the syntactic states determined by input to the parser.

C.1 Software Engineering Environment Classes

```
/* Interface for the ISLE SEE classes used to represent a program in
an OODB. */

#ifndef ISEEC_H
#define ISEEC_H
#include <time.h>
#include <sys/stdtypes.h>

#include <id_oodb_externs.hh>
#include <id_oodb.hh>
#include <id_symtab.h>
#include <id_type_expr.h>
#include <i_operator.h>

class Declaration {
 public:
  scope *local_scope;
  symtab_entry_list *decls;
  char  *src_code;

  /* Constructor for Declarations.  Takes a pointer to the top scope
   * of the Module, a collection of declarations, and an optional
   * pointer to the source code, if it is decided that the source be
   * needed to be stored in some manner.
   */
  Declaration(scope *ls, symtab_entry_list *d, char *sc = NULL);
  ~Declaration();
};


#define DEFINITION_MODULE        0
#define IMPLEMENTATION_MODULE    1
#define INTERFACE_MODULE    2

class id_Module {
 public:
```

```
 int module_type;
 char *module_name;
 id_oodb_list *id_Classes;
 id_oodb_list *imported_modules;
 Declaration *decl;
 long last_touch;
 int code_generation;
 int has_external_obj;

 /* Constructor for Module. The type and name of the module must be specified
  * Will automatically create an empty collection of classes and a
  * declaration object with the scope ls.
  */
 id_Module(int t, char *mn, scope *ls = NULL, int cg = 1);
 ~id_Module();
};


class id_Class {
 public:
 symtab_entry *se;
 symtab_entry_list *super_classes;
 type_expression_list *storage_type;
 id_oodb_list *methods;
 id_oodb_list *tell_methods;
 Declaration *decl;
 int is_forward;              /* is it a forward declaration? */
 int is_external;             /* is it an external declaration? */
 id_oodb_list *gknow;           /* class knowledge            */

 /* Constructor for id_Class. Only the symbol table entry need be
  * specified. Will automatically create an empty collection of methods,
  * and declaration object with scope ls.
  */
 id_Class(symtab_entry *i_se, symtab_entry_list *sc,
     type_expression_list *st, scope *ls, int is_f = 0,
     id_oodb_list *gk = NULL);
 ~id_Class();
```

```
};


/* Abstract syntax tree */

class ast_list : public id_oodb_list {
public:
  type_expression_list *convert_to_type_expr_list();
};


class ast_node {
 public:
  oper op;                          /* defined in oper.h */
  ast_node *obj_instance;           /* valid only for method invocations */
  symtab_entry *method;             /* valid only for method invocations */
  type_expression_list *which_type;
  union {
    char  *s;
    int   i;
    float f;
    char  c;
    ast_list *operand_list;
    symtab_entry *se;
  } semantic_value;

  /* Discriminator required by ObjectStore */
  int discriminant();


  /* Constructors for ast_node.  We need constructors for the different
   * types in the union.
   */

  ast_node (oper i_op, type_expression_list *wt);
  ast_node (oper i_op, type_expression_list *wt, char *i_s);
  ast_node (oper i_op, type_expression_list *wt, int i_i);
  ast_node (oper i_op, type_expression_list *wt, float i_f);
```

```
 ast_node (oper i_op, type_expression_list *wt, char i_c);
 ast_node (oper i_op, type_expression_list *wt, symtab_entry *i_se);
 ast_node (oper i_op, type_expression_list *wt, ast_node *obj_i,
     symtab_entry *m, ast_list *ol);
 ast_node (oper i_op, type_expression_list *wt, ast_list *ol);
 ~ast_node();
 /* deletes the type expression list and all the included type_expressions. */
 void del_which_type();

 /* Stuff required by ObjectStore */
private:
 int os_discriminator_value;
 void os_set_discriminator_value();

};

enum method_types { M_NOT_METHOD, M_ASK, M_TELL, M_QUERY, M_OPERATOR,
        M_CONSTRUCTOR, M_DESTRUCTOR };
typedef enum method_types method_types;

class id_Method {
public:
 symtab_entry *se;                    /* Entry in symbol table */
 method_types method_type;
 type_expression_list *parm_type;
 type_expression_list *storage_type;
 symtab_entry_list *parms;
 int is_public;
 symtab_entry *of_class;
 int body_declared;
 ast_node    *asf;                    /* Abstract Syntax Forest */
 Declaration  *decl;

 /* Constructor for Method. Only the symbol table entry need be specified.
  * Will automatically create an empty abstract syntax forest, and a
  * declaration object with the scope is. */
 id_Method(symtab_entry *i_se, method_types mt, type_expression_list *pt,
   type_expression_list *st, symtab_entry_list *fp, int p,
```

```
    symtab_entry *oc, int bd = 0, ast_node *a = NULL,
    scope *ls = NULL);
  ~id_Method();
};

#endif
```

C.2 The Type Expression Class

```
/* Class definition for the type_expression class used by the symbol table,
 * and for static semantic checking.  Must include "id_oodb.h" and
 * "id_oodb_externs.h", the generic object-oriented database interface
 *  before including this file.
 */

#ifndef TYPE_EXPRESSION_H
#define TYPE_EXPRESSION_H

#include <id_oodb_externs.hh>
#include <id_oodb.hh>

extern class symtab_entry;

enum atomic_type { T_NULL, T_ENUM, T_INT, T_REAL, T_BOOLEAN, T_STRING,
T_CHAR,
                T_ADDRESS, T_ARRAY, T_OBJECT, T_LIST, T_VARIABLE,
          T_PREDICATE, T_ARITHMETIC};
typedef enum atomic_type atomic_type;

class type_expression {
 public:
  int linenumber;
  atomic_type id_type;
  union {
    int size;                    /* valid only for arrays              */
    symtab_entry *id_class;      /* valid only of objects and enum types */
  } u;
```

```
  /* These are the constructors for the type_expression. */
  type_expression(int ln, atomic_type t);
  type_expression(int ln, atomic_type t, int s=0);
  type_expression(int ln, atomic_type t, symtab_entry *se=NULL);

  /* Discriminator required by ObjectStore */
  int discriminant();

  /* overloaded operators */
  int operator==(type_expression& t);
};

class type_expression_list : public id_oodb_list {

 public:
  /* Given that tel and "this" are valid type_expression_lists, verifies
   * that tel matches. Returns a 1 when it matches and a 0 when
   * it doesn't.
   */
  int compare_type(type_expression_list *tel);

  /* Given that tel is a valid type_expression_list, returns a copy
   * of it.
   */
  type_expression_list *make_copy(type_expression_list *tel);

  /* Given that tel is a valid type_expression_list, appends its contents
   * to this list. The original list is unchanged.
   */
  type_expression_list *append(type_expression_list *tel);

  /* Returns the only type_expression if type_expression_list is singleton,
   *  NULL otherwise.
   */
  type_expression *singleton();

  /* Returns a 1 if the list is empty, 0 otherwise */
  int empty();
```

```
  /* Deletes elements of the list, preserving only those type_expressions
   * whose atomic_type is t
   */
  void destroy_elements(atomic_type t = T_OBJECT);

  /* Prints a list */
  void printf_elements();

};

#endif
```

C.3 The Symbol Table Class

```
/* Class definition for the symbol table object and related sub-objects used
 * in the IMPORT/DOME compiler.
 */

#ifndef SYMTAB_H
#define SYMTAB_H

#include <id_type_expr.h>
extern class ast_node;
extern class ast_list;
extern class Method;
extern class id_Class;

/* Using the predifined list type for symtab_entry_list and block_list. */
class symtab_entry_list : public id_oodb_list {
 public:

  /* Compares the names in this list with the names in sel.  Returns 1 if
   * they are the same, 0 of not.
   */
  int compare_names(symtab_entry_list *sel);

  /* Deletes the elements of the list */
```

```
 void destroy_elements();

 /* Prints a list */
 void printf_elements();
};


/* This class defines a scope within a symbol_table. It contains a
 * pointer to the previous scope, the current scoping depth, and a local
 * table for identifiers declared in this scope.
 */

#define SIZE_OF_SCOPE 53    /* Hash table size within scope */

class scope {
 public:
 scope *previous;
 int depth;
 symtab_entry_list *decls;
 symtab_entry_list *local_symtab[SIZE_OF_SCOPE];

 /* Constructor for scope. The scope, p, is the previous scope.
  * The new scope will have previous point to p and depth set to
  * p->depth + 1.
  */
 scope(scope *p);

 /* Constructor for scope. The previous pointer and depth will be
  * set to 0.
  */
 scope();

 /* Prints out the entries in the scope. */
 void printf_elements();

 /* Destroys the symbol table_entries associated with this scope.
  * Also destroys the list decls.
  */
```

```
 void destroy_elements();
};


/* This class defines a symbol table entry, consisting the name as seen
 * in the source, a generated name that is used in the C++ translation,
 * a type, and a pointer to a type expression or to a scope, if the entry
 * is for an object. The scope type is used as a convenience for semantic
 * checking.
 */

enum symbol_types { S_CONST, S_TYPE, S_VARIABLE, S_PARAMETER,
S_CLASS,
          S_METHOD, S_DOME_SYMBOL };
typedef enum symbol_types symbol_types;
enum parm_io_types { P_NOT_PARM, P_IN, P_OUT, P_INOUT};
typedef enum parm_io_types parm_io_types;

class symtab_entry {

 public:
  char *symbol_name;
  char *generated_name;
  symbol_types symbol_type;
  ast_list *use_list;            /* contains a list of abstract syntax
                                  * trees where the variable appears. */
  int mark;                       /* used in searching      */

  union {
    type_expression_list *storage_type;  /* valid for variables, methods,
                  * type names and constants.    */
    id_Class          *obj;          /* valid for objects (classes) */
    Method            *method;       /* valid only for methods      */
  } u;

  parm_io_types         parm_io_type;  /* valid only for parameters     */
  ast_node             *const_value;   /* valid only for constants      */
```

```
int  is_public;              /* valid only for members of objects */
int  is_static;          /* valid for external C++ class methods */
int arity;                   /* valid only for dome              */
int is_prim;                 /* symbols                          */

/* These are the constructors for the symbol table entry. The generated
 * name is created by this method from the module name and the user's
 * symbol name.
 */

/* Used for variables, type names, formal parameters. */
symtab_entry(char *sn, symbol_types st, type_expression_list *st_t,
       int p=1, parm_io_types pit=P_NOT_PARM);

/* Used for methods. */
symtab_entry(char *sn, Method *m, int p=1);

/* Used for constants */
symtab_entry(char *sn, symbol_types st, ast_node *cv);

/* Used for classes (objects) */
symtab_entry(char *sn, symbol_types st, id_Class *i_obj);

/* Used for imported classes and types */
symtab_entry(char *sn, symtab_entry *i_entry);

/* Used for dome symbols  */
symtab_entry(char *sn, int ar, int ip);

/* Used to rename a generated name for EXTERNAL C++ calls
 * needed by INTERFACE MODULE
 */
void rename_genname(char *newgenname, int is_stat = 0);

/*  Discriminator required by ObjectStore */
int discriminant();
};
```

```
/* This class implements a symbol table for an object-oriented, lexically
 * scoped language.
 */
class Symtab {

public:
 scope *cur_scope;
 scope *cur_module;
 scope *global_scope;
 symtab_entry *cur_obj;

 /* Constructor for a symbol table.  Sets the current scope to the
  * global scope.
  */
 Symtab();

 /* Creates a new scope and sets the current scope to it. */
 void enter_new_scope();

 /* Sets the current scope to the scope applicable when defining methods
  * of a class.  Returns a 1 if successful, 0 if the class has not been
  * defined in the symbol table.
  * Exceptions: Will give an error if not currently in the module scope.
  */
 int enter_obj_scope(char *symbol);

 /* Sets the current scope to the scope that was surrounding it.
  * Exceptions:  Will give an error on an attempt to leave a global
  *              scope or when the previous scope was NULL.
  */
 void leave_scope();

 /* Inserts a local or member variable into the symbol table if it
  * does not already exist.  If it already exists, it returns
  * a NULL, otherwise it returns a pointer to the symtab_entry.
  */
 symtab_entry *insert_var(char *symbol, type_expression_list *t);
```

```
/* Inserts a constant into the symbol table if it
 * does not already exist.  If it already exists, it returns
 * a NULL, otherwise it returns a pointer to the symtab_entry.
 */
symtab_entry *insert_const(char *symbol, ast_node *a);

/* Inserts a type name into the symbol table if it
 * does not already exist.  If it already exists, it returns
 * a NULL, otherwise it returns a pointer to the symtab_entry.
 */
symtab_entry *insert_type(char *symbol, type_expression_list *t);

/* Inserts an class name into the module scope; creates a new
 * scope for the members of the class and sets the current
 * scope to it.
 * Returns a NULL if the class already exists, or if not in the
 * module scope.  Returns a pointer to the symtab_entry otherwise.
 */
symtab_entry *insert_obj(char *symbol);

/* Inserts a method name into the symbol table under the current
 * class being defined.  Parameters are the name of the method, the
 * Method object and whether the method is public. Returns a NULL if
 * the symbol is already defined. Returns the symtab_entry otherwise.
 */
 symtab_entry *insert_method(char *symbol, Method *m, int p=1);

/* Inserts an imported item into the symbol table under the current
 * module being defined.  Takes the name of the item in this module
 * and a symbol table entry from the imported module.  Returns the
 * symtab entry created, NULL if the entry already exists.
 */
symtab_entry *insert_imported(char *symbol, symtab_entry *imported_entry);

/* Inserts a dome symbol entry into the current scope. */
symtab_entry *insert_dome_symbol(symtab_entry *se);
```

```
/* Returns a pointer to a symbol table entry if the symbol is a
 * method or a field in the object. Returns a NULL if it is not
 * found.
 */
symtab_entry *is_in_obj(char *symbol, symtab_entry *object_entry,
                        int *err_code = NULL,
                        type_expression_list *parm_list = NULL);

/* Returns a pointer to a symbol table entry if the symbol is a
 * superclass of object. Returns a NULL if it is not
 * found.
 */
symtab_entry *is_a_superclass(char *symbol, symtab_entry *object_entry);


/* Returns a pointer to the symbol table entry if the symbol is
 * defined in the current context. Returns a NULL if it is not
 * found.
 */
symtab_entry *is_in(char *symbol);

/* Returns a pointer to the symbol table entry if the symbol is
 * defined in the immediate scope. Returns a NULL if it is not
 * found.
 */
symtab_entry *is_in_local(char *symbol);

/* Returns a pointer to the symbol table entry if the symbol is
 * defined in the outermost (module) context. Returns a NULL if it
 * is not found. This is mainly an optimisation feature; classes
 * can only be defined in the module level, and if we are looking for
 * a class definition, it can only be at the module level.
 */
symtab_entry *is_in_module(char *symbol);

/* Returns a pointer to the symbol table entry if it is found in the
 * scope sc, NULL otherwise. (CEH must check parms for methods)
 */
```

```
 symtab_entry *is_in_scope(char *symbol, scope *sc,
                           type_expression_list *parm_list=NULL );

 /* Returns a pointer to the symbol table entry if there is a name
  * match.  Does not check signature.  Returns NULL if no match.
  */
 symtab_entry *is_name_in_scope(char *symbol, scope *sc);
};

extern Symtab *symbol_table, *lib_symbol_table;

#endif
```

C.4 The Semantic Action Controller Class

```
/* Interface for the semantics/object builder for the IMPORT language */


#ifndef SEMANTICS_H
#define SEMANTICS_H

#include <id_seec.h>
#include <id_type_expr.h>
#include <id_symtab.h>

class id_semantic_controller {
 public:
  Module *cur_module;
  parm_io_types cur_parm_io_type;
  int    is_public;
  int    is_overridden;
  Module *imported_module;
  Method *cur_method;
  type_expression_list *cur_parm_type;
  symtab_entry_list *cur_parm_names;
  id_oodb_list *cur_methods;
  id_oodb_list *cur_classes;
  id_oodb_list *cur_tell_methods;
```

```
/* Constructor for semantic controller */
id_semantic_controller();

/* Imports all entries in the Module m into the cur_module.  If m
 * is NULL, has no effect.
 */
void import_Module(int ln, Module *m);

/* Builds a module object and sets the cur_module to it.  Takes a module
 * type and an optional local scope.  A new scope will be created for the
 * module, and the cur_scope of the symbol_table will be set to this.  The
 * cur_module of the symbol_table will also be set to this scope value.
 * The cur_module of the semantic_cont will be set to the module object.
 */
void build_Module(int ln, int t, char *mn, scope *ls = NULL);

/*  Builds user defined  types and stores in symbol table
 *  Side effect is that it destroys iden.
 */

void build_user_type(int ln, char * iden, type_expression_list *tel);

/* Builds a symtab entry for the variable iden of type te and inserts
 * it into the symbol table.
 * Side effect: frees the iden string.
 */
void build_var(int ln, char *iden, type_expression_list *te);

/* Builds a symtab entry for the constant with value v and inserts
 * it into the symbol table.
 * Side effect: frees the iden string.
 */
void build_const(int ln, char *iden, ast_node *v);

/* Builds a singleton type_expression_list or inserts a type_expression
 * of atomic_type t into list tel.  s is the optional size argument in
 * the case of declaring arrays. ln is the line number of the declaration.
```

```
 * The second version is for classes. c is the class
 */
type_expression_list *build_type(int ln, atomic_type t,
                                 type_expression_list *tel=NULL, int s=0);

type_expression_list *build_type(int ln, atomic_type t, symtab_entry *c);

/* Determines if the class or type named iden exits, and if so, returns its
 * storage_type.
 */
type_expression_list *get_type(int ln, char *iden);

/* Builds an ast_node with operator op, and a type_expression with atomic
type
 * a. ln is the line number of the expression.
 */
ast_node *build_arith_ast(int ln, oper op, atomic_type a, char *datavalue);
ast_node *build_arith_ast(int ln, oper op, atomic_type a, int datavalue);
ast_node *build_arith_ast(int ln, oper op, atomic_type a, float datavalue);
ast_node *build_arith_ast(int ln, oper op, atomic_type a, char datavalue);
ast_node *build_arith_ast(int ln, oper op, atomic_type a,
          symtab_entry *datavalue);

/* Builds an ast_node with operator op, and operand_list made up of the
 * operands.
 */
ast_node *build_unary_ast(int ln, oper unary_op, ast_node *operand);
ast_node *build_arith_ast(int ln, oper binary_op, char *string_rep,
         ast_node *left_operand, ast_node *right_operand);


/* Builds an ast_node for an identifier (local variable or object member)
 * as a location for data. The second version handles array references.
 */
ast_node *build_loc_ast(int ln, char *string_rep,
         symtab_entry *inherited_class = NULL);
ast_node *build_loc_ast(int ln, ast_node *base, ast_node *index);
```

```
/* Builds an ast_node for an assignment_statement                */
ast_node *build_assign_ast(int ln, oper assign_op, char *string_rep,
            ast_node *loc, ast_node *expr);

/* Builds an ast_node for AND and OR                             */
ast_node *build_boolean_ast(int ln, oper boolean_op,
            ast_node *left_operand, ast_node *right_operand);

/* Builds an ast_node for relations                              */
ast_node *build_relation_ast(int ln, oper rel_op, char *string_rep,
            ast_node *left_operand, ast_node *right_operand);

/* Builds an ast_node for statement_lists. If v is NULL, creates a new
 * ast_node, otherwise, it adds the ast_node for the statement s to the
 * statement_list result v.
 */
ast_node *build_stmt_list(ast_node *s, ast_node *v);

/* Builds up an IF statement and from its component parts. If v is NULL,
 * creates a new ast_node, and in this case, opd must contain the ast_node
 * for the expression. After the expression is inserted, the then clause,
 * and the elsif clauses must be inserted in order, and finally the
 * else clause.
 */
ast_node *build_if_stmt(int ln, ast_node *opd, ast_node *v);

/* Builds an elsif clause. */
ast_node *build_elsif(int ln, ast_node *expr, ast_node *stmt);

/* Builds an else clause */
ast_node *build_else(ast_node *stmt);

/* Builds a return statement, the expression expr is optional. */
ast_node *build_return(int ln, ast_node *expr = NULL);

/* Builds a while statement. */
ast_node *build_while(int ln, ast_node *expr, ast_node *stmt);
```

```
/* Builds a repeat statement. */
ast_node *build_repeat(int ln, ast_node *stmt, ast_node *expr);

/* Builds a loop statement. */
ast_node *build_loop(ast_node *stmt);

/* Builds an exit statement. */
ast_node *build_exit();

/* Builds an halt statement. */
ast_node *build_halt();

/* Builds a for statement. The arguments are linenumber, index
 * variable, initial value, final value, up or down (up = 1, down =
 * 0), statement list, and optional by clause
 */
ast_node *build_for(int ln, ast_node *idx, ast_node *i_val,
        ast_node *f_val, int up, ast_node *stmt,
        ast_node *by_exp = NULL);

/* Builds up an CASE statement and from its component parts. If v is NULL,
 * creates a new ast_node, and in this case, opd must contain the ast_node
 * for the expression. After the expression is inserted, the WHEN clauses,
 * and the OTHERWISE clause must be inserted in order.
 */
ast_node *build_case(int ln, ast_node *opd, ast_node *v);

/* Builds the WHEN and OTHERWISE clauses for the CASE statement.
cl is
 * the case label for the WHEN clause.
 */
ast_node *build_case_clause(oper op, ast_node *stmt, ast_node *cl = NULL);

/* Builds a block statement. */
ast_node *build_block(ast_node *stmt);

/* Builds a method named iden as a member of the current object */
Method *build_method(int ln, char *iden, method_types mt,
```

```
        type_expression_list *return_type=NULL);

  /* provides for renaming of external C++ methods in the INTERFACE mod-
ule */
  void rename_ext_method(int ln, char *iden,  char *newname, int is_stat =
0);

  /* provides for renaming of external C++ fields in the INTERFACE mod-
ule */
  void rename_ext_field(int ln, char *iden,  char *newname);

  /* Builds a class named iden and sets cur_obj in the symbol table to
   * the new id_class obj created.
   */
  id_Class *build_class(int ln, char *iden, symtab_entry_list *sc = NULL,
        int is_f = 0);


  /* Builds a list of symbol table entries of superclasses of a class.
   * If v is NULL, creates a new list and returns the new list as the
   * result.
   */
  symtab_entry_list *build_superclasses(int ln, char *superclass_id,
              symtab_entry_list* v = NULL);

  /* Sets up the symbol table and the semantic controller for parsing
   * methods. Handles both IMPLEMENTATION and INTERFACE methods
   * the cname paramater is for renaming DEFINITION MODULE class names
   * for C++ compatability.
   */
  void setup_methods(int ln, int mod_type, char *iden, char *cname=NULL);

  /* Associates an method body with the prototype. */
  void fill_in_method(Method *m, ast_node *i_asf);

  /* Builds a list of arguments. If v is NULL creates a new ast_list
   * and inserts the first argment and returns it. Otherwise, inserts
   * the arguments in v and returns it.
```

```
 */
ast_list *build_arg_list(int ln, ast_node *expr, ast_list *v);

/* Sets up the inherited class variable for method invocations. */
symtab_entry *setup_inherited(int ln, char *inherited_class_iden);

/* Builds a method invocation.  If the method has no arguments a NULL
 * is given as the argument to this procedure.
 */
ast_node *build_meth_inv(int ln, char *inherited_class,
          method_types mt, ast_node *obj_expr,
          char *method_name, ast_list *args = NULL);

/* Checks that a method invocation is a valid expression */
void check_meth_inv(int ln, ast_node *meth_inv);

/* Builds a terminate statement. */
ast_node *build_terminate();

/* Builds an INTERRUPT clause */
ast_node *build_on_interrupt(ast_node *stmts);

/* Builds WAIT DURATION statement */
ast_node *build_wait_dur(int ln, ast_node *expr, ast_node *stmts,
          ast_node *int_clause=NULL);

/* Builds WAIT FOR statement */
ast_node *build_wait_for(int ln, ast_node *obj_expr, char *method_name,
          ast_list *args, ast_node *stmts,
          ast_node *int_clause = NULL,
          char *inherited_class = NULL);

/* Builds WAIT FOR TRIGGER statement */
ast_node *build_wait_for_trigger(int ln, ast_node *obj_expr,
              ast_node *stmts,
              ast_node *int_clause = NULL);
```

```
/* Builds interrupt trigger statement
ast_node *build_interrupt_trigger(int ln, ast_node *obj_expr,
            ast_node *meth_name);
 */

/* Builds a key for an application */
void build_key(int ln, char *key_name, char *module_name, char *class_name,
     char *method_name, ast_list *args);


/* Retrieves a definition module with the module_name and returns it.
 * If the module is not found, NULL is returned.
 */
Module *retrieve_module(int ln, char *module_name);

/* Sets up the modules and for import */
void setup_build_import(int ln, char *module_name);

/* Builds symbol table entries for imported items */
void build_import(int ln, char *source_iden, char *target_iden = NULL);


/* BUILT-IN FUNCTIONS */

ast_node *build_abs(int ln, oper ir_op, ast_node* arg);

ast_node *build_cap(int ln, oper char_op, ast_node* arg);

ast_node *build_chartostr(int ln, oper char_op, ast_node* arg);

ast_node *build_chr(int ln, oper int_op, ast_node* arg);

ast_node *build_float(int ln, oper int_op, ast_node* arg);

ast_node *build_inttostr(int ln, oper int_op, ast_node* arg);

ast_node *build_lower(int ln, oper str_op,ast_node* arg);
```

```
ast_node *build_max(int ln, oper scalar_op, ast_node* arg);

ast_node *build_maxof(int ln, oper scale_op,ast_list* arg);

ast_node *build_min(int ln, oper scalar_op, ast_node* arg);

ast_node *build_minof(int ln, oper scale_op,ast_list* arg);

ast_node *build_odd (int ln, oper scalar_op,ast_node* arg);

ast_node *build_ord(int ln, oper ord_op,ast_node* arg);

ast_node *build_persistent(int ln, oper op,ast_node* arg);

ast_node *build_position(int ln, oper str_op,ast_node* arg1,
        ast_node* arg2);

ast_node *build_realtostr(int ln, oper r_op, ast_node* arg);

ast_node *build_round(int ln, oper real_op, ast_node* arg);

ast_node *build_schar(int ln, oper str_op,ast_node* arg1,
        ast_node* arg2);

ast_node *id_semantic_controller::build_simtime(int ln);

ast_node *build_strcat(int ln, oper op,ast_list* arg);

ast_node *build_strlen(int ln, oper str_op, ast_node *arg);

ast_node *build_strtoint(int ln, oper str_op, ast_node* arg);

ast_node *build_strtoreal(int ln, oper str_op, ast_node* arg);

ast_node *build_substr(int ln, oper str_op,ast_node* arg1,
        ast_node* arg2,ast_node* arg3);

ast_node *build_trunc(int ln, oper real_op,ast_node* arg);
```

```
ast_node *build_upper(int ln, oper str_op, ast_node* arg);

ast_node *build_val(int ln, oper ord_op,ast_node* arg1, ast_node* arg2);

/* BUILT-IN PROCEDURES */

ast_node *build_dec(int ln, oper op,ast_node* arg);

ast_node *build_dec(int ln, oper op,ast_node* arg1, ast_node* arg2);

ast_node *build_dispose(int ln, oper op, ast_node* arg);

ast_node *build_inc(int ln, oper op,ast_node* arg);

ast_node *build_inc(int ln, oper op,ast_node* arg1, ast_node* arg2);

ast_node *build_input(int ln, oper scale_op, ast_list* arg);

ast_node *build_insert(int ln, oper str_op,ast_node* arg1,
        ast_node* arg2,ast_node* arg3);

ast_node *build_output(int ln, oper scale_op, ast_list* arg);

/* Builds an ast_node for making a persistent object */
ast_node *build_pnew(int ln, ast_node* arg1, ast_node* arg2,
        ast_list* args=NULL);

ast_node *build_replace(int ln, oper str_op,ast_node* arg1,
        ast_node* arg2,ast_node* arg3,ast_node* arg4);

ast_node *build_strtochar(int ln, oper op,ast_node* arg1, ast_node* arg2);

/* Builds an ast_node for making a transient object */
ast_node *build_tnew(int ln, ast_node *loc, ast_list *args=NULL);

/* Builds interrupt statement */
ast_node *build_interrupt(int ln, oper i_op, ast_node *obj_expr,
```

```
                ast_node *meth_name);

  /* END BUILD-INS */

 private:
  /* Internal error printer */
  void internal_error(int code);
  /* Semantic Error Message Printer */
  void semantic_error(int lineno, int code, char *message = NULL);

};

extern id_semantic_controller *semantic_cont;

#endif
```

Appendix D

Generic Object Oriented Database Interface Specification

D.1 Introduction

This appendix contains the C++ classes defining the Generic Object-Oriented Database (GOOD) interface developed for IMPORT/DOME. The clases listed here constituite our internal **release 4.1** of the GOOD interface. The GOOD was developed to insulate implementation at the persistent object storage level from any particular database and to premit rapid porting of IMPORT/DOME to other database hosts. This interface is implemented for ObjectStore [7] and we are currently developing an implementation for the DARPA OpenDB [37].

D.2 Database

A database provides a way to manipulate objects in persistent storage and is organized as an Unix-like directory hierachy. All persistent objects in the database must have a named entry-point into the database which is done by DNEW macro. They then can be manipulated through the directory objects (see directory class below) where they reside. An open count is maintained on the number of times a database has been opened. When the open count is decreased to 0, the database is closed. All databases are automatically closed when the program terminates.

id_oodb_database::root_dir()
id_oodb_directory *root_dir();
Returns a pointer to the root directory in this database.

id_oodb_database::close()
void close();
Decrements the open count by one if the resulting open count is greater than one. If the open count is zero the database is closed. If called from within a transaction, the above does not take place until the end of the transaction.

id_oodb_database::create()
static id_oodb_database *create (char *pathname, int mode = 0664,
int if_exists_overwrite = 0);
Creates a database with the specified path name and mode. The database is also opened for reading and writing and its open count is incremented.

id_oodb_database::get_path_name()
char *get_pathname();
Returns the pathname of the target database.

id_oodb_database::is_open()
int is_open();
Returns 1(true) if the target database is open, 0(false) otherwise.

id_oodb_database::is_writable()
int is_writable();
Returns 1(true) if the database is writable by the current process. Returns 0(false) if the database has been opened for read-only or if the process does not have write permission.

id_oodb_database::lookup()
static id_oodb_database *lookup (const char *pathname, int create_mode = 0);
Returns a pointer to the database with the specified pathname, but does not open it. Returns 0 if the database is not found.

id_oodb_database::of()

static id_oodb_database *of (void *item);
Returns a pointer to the database in which the object item resides.

id_oodb_database::open()
void open (int read_only);
Increments the open count of the database and opens it for the specified access type - read-only if 1(true) or read-write if 0(false).

static id_oodb_database *open (const char *path_name, int read_only = 0);
Increments the open count of the specified database "path_name". Sets up the access type specified by "read_only" and returns a pointer to that database.

id_oodb_database::operator ==()
int operator ==(id_oodb_database *that);
Returns 1(true) if the target database is same as the database pointed to by "that".

id_oodb_database::allow_external_pointers()
void allow_external_pointers();
Extend the validity of cross-database pointers. After this function is called, cross-database pointers can be stored in the current process and subsequent processes.

id_oodb_database::destroy()
void destroy ();
Deletes the database for which this function is called. Must be called inside a transaction.

id_oodb_database::create_segmemt()
id_oodb_segment *create_segment();
Returns a pointer to the newly created segment in the specified database.

D.3 Directory

The named objects in a database are organized in an Unix-like directory hierarchy. A directory may contain other directories and named objects. The default working directory is the root directory ("/") which is created automatically whenever a new database is created. Path names to other directories or named objects are specified relative to the current working directory or by using the complete path name starting with the root.

id_oodb_directory::lookup()
static void *lookup (id_oodb_database *db, const char *path_name);
Looks up a named object, which can be either an ordinary object or a directory object, in the database. A pointer to the object is returned.

void *lookup (const char *obj_name);
Looks up an object by the object's string name in this directory and returns a pointer to that object.

id_oodb_directory::insert()
static void insert (const char *dir, void *item, const char *name);
Inserts the object pointed to by "item" into the directory "dir", with the name "name".

void insert (void *item, const char *obj_name);
Inserts the object pointed to by item into this directory and names it with the specified name "obj_name".

id_oodb_directory::ls()
void ls ();
Returns a list containing the names of the named objects and subdirectories in this directory.

id_oodb_directory::remove()
int remove (const char *obj_name);
Removes an entry for a named object "obj_name" from this directory. Does not delete the object from the database. Returns 1 if the object has been removed successfully, else 0.

id_oodb_directory::mkdir()

id_oodb_directory *mkdir (const char *dir_name);
Makes a new subdirectory "dir_name" in this directory, and returns a pointer to this new subdirectory.

id_oodb_directory::rmdir()
int rmdir (const char *dir_name);
Removes an entry for a named subdirectory "dir_name" from this directory. Does not delete the subdirectory from the database. Returns 1 if the subdirectory has been removed successfully, else 0.

id_oodb_directory::getdir()
static id_oodb_directory *getdir (id_oodb_database *db, const char *dir_name);
Gets a directory object with its entry name "dir_name" in the specified database. Returns a pointer to the directory if found, else returns a NULL pointer.

D.4 Lists and Cursors

D.4.1 List

id_oodb_list::create()
static id_oodb_list *create (id_oodb_database *db);
Creates a list in the specified database and returns a pointer to the created list.

static id_oodb_list *create (id_oodb_configuration *conf);
Creates a list in the specified configuration and returns a pointer to the created list.

id_oodb_list::insert()
void insert (void *item);
Inserts the passed in item at the end of the list.

id_oodb_list::insert_first()
void insert_first (void *item);
Inserts the passed in item at the front of the list.

id_oodb_list::remove()
void remove (void *item);
Removes the specified item from the list.

id_oodb_list::remove_first()
void remove_first();
Removes the first element from the list.

id_oodb_list::remove_last()
void remove_last();
Removes the last element from the list.

id_oodb_list::size()
int size();
Returns the number of elements in the list.

D.4.2 List Cursor

A list_cursor is created for a particular list which is specified when the list_cursor is created.

id_oodb_list_cursor::first()
void *first();
Sets the cursor at the first element of the list and returns a pointer to the element.

id_oodb_list_cursor()
id_oodb_list_cursor (id_oodb_list *list);
Constructor - creates a list_cursor for the specified list.

id_oodb_list_cursor::insert_after()
void *insert_after(void *item);
Inserts the passed in object after the current element in the list.

id_oodb_list_cursor::insert_before()
void *insert_before(void *item);
Inserts the passed in object before the current element in the list.

id_oodb_list_cursor::is_null()
int is_null();
Returns 0(false) if the cursor is located at a valid element of the list; returns 1(true) otherwise.

id_oodb_list_cursor::last()
void *last();
Positions the cursor at the last element of the list and returns a pointer to the element.

id_oodb_list_cursor::more()
int more();
Returns 1(true) if the cursor is not at the last element of the list; returns 0(false) otherwise.

id_oodb_list_cursor::next()
void *next();
Advances the cursor to the next element of the list and returns a pointer to the element.

id_oodb_list_cursor::previous()
void *previous();
Moves the cursor to the previous element of the list and returns a pointer to the element.

id_oodb_list_cursor::remove_at()
void remove_at();
Removes from the list the element at which the cursor is currently located.

id_oodb_list_cursor::retrieve()
void *retrieve();
Returns a pointer to the element of the list at which the cursor is currently located.

D.4.3 Parse List

id_oodb_parse_list::append()
void append (id_oodb_parse_list *l);
Appends the passed in list to the end of the target list. This is done by physically attaching the passed in list to the end of the target list and not by making a copy of the passed in list. Hence, changes made to one list may affect the other.

id_oodb_parse_list::create()
static id_oodb_parse_list *create (id_oodb_database *db);
Creates a list in the specified database and returns a pointer to the created list.

id_oodb_parse_list::insert()
void insert (void *item);
Inserts the passed in item at the end of the list.

id_oodb_parse_list::insert_first()
void insert_first (void *item);
Inserts the passed in item at the front of the list.

id_oodb_parse_list::remove()
void remove (void *item);
Removes the specified item from the list.

id_oodb_parse_list::remove_first()
void remove_first();
Removes the first element from the list.

id_oodb_parse_list::remove_last()
void remove_last();
Removes the last element from the list.

D.4.4 Parse List Cursor

A list_cursor is created for a particular list which is specified when the list_cursor is created.

id_oodb_parse_list_cursor::first()
void *first();
Sets the cursor at the first element of the list and returns a pointer to the element.

id_oodb_parse_list_cursor()
id_oodb_parse_list_cursor (id_oodb_parse_list *list);
Constructor - creates a list_cursor for the specified list.

id_oodb_parse_list_cursor::insert_after()
void insert_after(void *item);
Inserts the passed in object after the current element in the list.

id_oodb_parse_list_cursor::is_null()
int is_null();
Returns 0(false) if the cursor is located at a valid element of the list; returns 1(true) otherwise.

id_oodb_parse_list_cursor::more()
int more();
Returns 1(true) if the cursor is not at the last element of the list; returns 0(false) otherwise.

id_oodb_parse_list_cursor::next()
void *next();
Advances the cursor to the next element of the list and returns a pointer to the element.

id_oodb_parse_list_cursor::remove_at()
void remove_at();
Removes from the list the element at which the cursor is currently located.

id_oodb_parse_list_cursor::retrieve()
void *retrieve();

Returns a pointer to the element of the list at which the cursor is currently located.

id_oodb_parse_list_cursor::tail()
id_oodb_parse_list *tail();
Returns the tail of the list starting from the current location of the cursor.

D.4.5 Set

id_oodb_set::create()
static id_oodb_set *create (id_oodb_database *db);
Creates a set in the specified database and returns a pointer to the created set.

static id_oodb_set *create (id_oodb_configuration *conf);
Creates a set in the specified configuration and returns a pointer to the created set.

id_oodb_set::is_empty()
int is_empty ();
Returns 1(true) if the set is empty 0(false) otherwise.

id_oodb_set::insert()
void insert (void *item);
Inserts the passed in item into the set.

id_oodb_set::remove()
void remove(void *item);
Removes the specified item from the set.

id_oodb_set::size()
int size();
Returns the number of elements in the set.

D.4.6 Set Cursor

A set_cursor is created for a particular set which is specified when the set_cursor is created.

id_oodb_set_cursor::first()
void *first();
Sets the cursor at the first element of the set and returns a pointer to the element.

id_oodb_set_cursor()
id_oodb_set_cursor (id_oodb_set *set);
Constructor - creates a set_cursor for the specified set.

id_oodb_set_cursor::is_null()
int is_null();
Returns 0(false) if the cursor is located at a valid element of the set; returns 1(true) otherwise.

id_oodb_set_cursor::more()
int more();
Returns 1(true) if the cursor is not at the last element of the set; returns 0(false) otherwise.

id_oodb_set_cursor::next()
void *next();
Advances the cursor to the next element of the set and returns a pointer to the element.

id_oodb_set_cursor::remove_at()
void remove_at();
Removes from the set the element at which the cursor is currently located.

id_oodb_set_cursor::retrieve()
void *retrieve();
Returns a pointer to the element of the set at which the cursor is currently located.

D.4.7 Tree

id_oodb_tree::create()
static id_oodb_tree *create (id_oodb_database *db, void *root);
Creates a tree in the specified database, with the object pointed to by the argument root as the root of the tree. Returns a pointer to the created tree.

id_oodb_tree::is_empty()
void is_empty ();
Returns 1(true) if the tree is empty; returns 0(false) otherwise.

D.4.8 Tree Cursor

A tree_cursor is created for a particular tree which is specified when the tree_cursor is created.

id_oodb_tree_cursor::add_child()
void add_child(void *item);
Adds the passed in item as a child of the current node.

id_oodb_tree_cursor::first_child()
void *first_child();
Advances the cursor to the first child of the current node and returns a pointer to the object stored therein.

id_oodb_tree_cursor()
id_oodb_tree_cursor (id_oodb_tree *tree);
Constructor - creates a tree_cursor for the specified tree.

id_oodb_tree_cursor::is_first_sibling()
int is_first_sibling();
Returns true (1) if the current node is the first sibling, false (0) otherwise.

id_oodb_tree_cursor::is_last_sibling()
int is_last_sibling();
Returns true (1) if the current node is the last sibling, false (0) otherwise.

id_oodb_tree_cursor::is_leaf()
int is_leaf();
Returns true (1) if the current node is a leaf, false (0) otherwise.

id_oodb_tree_cursor::is_root()
int is_root();
Returns true (1) if the current node is the root of the tree, false (0) otherwise.

id_oodb_tree_cursor::next_sibling()
void *next_sibling();
Advances the cursor to the next sibling of the current node, if any, and returns a pointer to the object stored therein.

id_oodb_tree_cursor::parent()
void *parent();
Moves the cursor to the parent of the current node and returns a pointer to the object stored therein.

id_oodb_tree_cursor::prev_sibling()
void *prev_sibling();
Advances the cursor to the previous sibling of the current node, if any, and returns a pointer to the object stored therein.

id_oodb_tree_cursor::remove_at()
void remove_at();
Removes the current node from the tree, if it does not have any children. Does nothing otherwise.

id_oodb_tree_cursor::root()
void *root();
Sets the cursor at the root of the tree and returns a pointer to the element.

id_oodb_tree_cursor::retrieve()
void *retrieve();
Returns a pointer to the object stored in the current node of the tree.

D.5 Workspace

id_oodb_workspace::create()
static id_oodb_workspace *create(id_oodb_database *db, const char *name, id_oodb_workspace *parent=0);
Constructs a workspace called "name", which can not be omitted, with the specified parent "parent". If no parent is specified, the new workspace is made a child of the current workspace. Returns a pointer to the created workspace.

id_oodb_workspace::id_oodb_workspace()
id_oodb_workspace(id_oodb_database *db, const char *name, id_oodb_workspace *parent=0);
This constructor is essentially the same as id_oodb_workspace::create(). It constructs a workspace called "name" with the specified parent "parent" in the specified database "db". If no parent is specified, the new workspace is made a child of the current workspace.

id_oodb_workspace::create_global()
static id_oodb_workspace *create_global(id_oodb_database *db, const char *name);
Creates a global workspace with the specified name which can not be omitted in the specified database "db". A pointer to the created workspace is returned.

id_oodb_workspace::current()
static id_oodb_workspace *current();
Returns the current workspace, which is the workspace on which the most recent invocation of set_current() has been made. If no invocation of set_current has been made in the current process, then the global workspace is returned as the current workspace.

id_oodb_workspace::get_name()
const char *get_name();
Returns the name of the current workspace.

id_oodb_workspace::get_parent()
id_oodb_workspace *get_parent();
Returns a pointer to the parent workspace of the current workspace, or 0 if

the current workspace is the global workspace.

id_oodb_workspace::of()
static id_oodb_workspace *of(void *item);
Returns a pointer to the workspace that maps the object pointed to by item. Returns the current workspace if item is a non versioned object.

id_oodb_workspace::resolve()
void *resolve(id_oodb_database *db, const char *path_name, void *item);
Returns a pointer to the version of the object, pointed to by "item", which is made visible by the specified workspace "path_name".

void *resolve(void *item);
Returns a pointer to the version of the object that is made visible by this workspace.

id_oodb_workspace::set_current()
static void set_current(id_oodb_workspace *a_ws);
Sets the workspace "a_ws" as the current workspace. This action takes effect only at the beginning of the next transaction.

id_oodb_workspace::lookup()
static id_oodb_workspace *lookup(id_oodb_database *db, const char *path_name);
Returns a pointer to the workspace "path_name" in the specified database "db" if it is found, otherwise returns a NULL pointer.

D.6 Configuration

id_oodb_configuration::checkin()
void checkin (id_oodb_database *db, const char *name, int recursive = 1);
Removes the configuration "name" from the current workspace in the specified database "db", freezes it and inserts it in the parent workspace. The configuration is also made current for the parent on the branch that contains it.

id_oodb_configuration::checkout()

void checkout (id_oodb_database *db, const char *name, int recursive = 1);
Creates a new version of the target configuration "name" and inserts the new version into the current workspace with write access in the specified database "db". The branch containing the new version is also made the current branch.

id_oodb_configuration::checkout_branch()
void checkout_branch(id_oodb_database *db, const char *name, char *branch-name=0, int recursive=1);
Creates a new version of the target configuration "name" and inserts the new version into the current workspace with write access in the specified database "db". Like checkout, the new version will not be the same branch as the current branch, instead it will be an alternative version.

id_oodb_configuration::create()
static id_oodb_configuration *create(id_oodb_database *db, const char *name);
Creates a configuration called "name" in the specified database and returns a pointer to it.

id_oodb_configuration::merge()
void merge (id_oodb_database *db, id_oodb_workspace *ws, char *name);
Creates a new version of the target configuration and makes that the successor of the target configuration and the configuration "name" which is in workspace "ws". That is, this function merges two versions of configurations, one in current workspace the other in "ws", into a new version which is the successor of both configurations.

id_oodb_configuration::predecessor()
id_oodb_configuration *predecessor(id_oodb_database *db, const char *name);
Returns a pointer to the predecessor version of the target configuration and uses the predecessor version.

id_oodb_configuration::successors()
id_oodb_list *successor(id_oodb_database *db, const char *name);
Returns a pointer to the successor version of the target configuration and uses the successor version.

id_oodb_configuration::resolve()
void *resolve(id_oodb_database *db, const char *name);

Returns the version of the object "name" that is mapped by the target configuration in the specified database "name".

id_oodb_configuration::lookup()
static id_oodb_configuration *lookup(id_oodb_database *db, const char *name);
Returns a pointer which points to configuration "name" in the specified database "db".

D.7 Segment

id_oodb_oodb_segment::create()
id_oodb_segment *create(const id_oodb_database *db);
Creates a segment in the specified database and returns a pointer to the created segment.

id_oodb_oodb_segment::database_of()
id_oodb_database *database_of();
Returns a pointer to the database in which the target segment resides.

id_oodb_oodb_segment::destroy()
void destroy();
Deletes the target segment from the database. This results in the destruction of all the data the segment contains. This action cannot be undone by aborting the transaction.

id_oodb_oodb_segment::of()
static id_oodb_segment *of (void *item);
Returns a pointer to the segment where the object pointed to by item resides.

D.8 Macro Defines

Several macros have been defined to ease the process of allocating and deallocating objects in persistent database. These macros include DNEW, CNEW,

SNEW, and DELETE. DNEW is used for allocating objects into database, CNEW is specifically used for allocating objects into the specified configuration in the database for versioning purpose. SNEW is used for allocating objects into the specified segment for clustering purpose. Delete is simply used for deleting objects in the database.

ID_OODB_TXN_BEGIN, ID_OODB_TXN_COMMIT, and ID_OODB_-TXN_ABORT are transaction related macros. The first macro starts a transaction and controls the access mode provided by the argument read_only. If the argument read_only is true (1), then the access mode is read only otherwise the access mode is read-write. The second one signals the transaction has been committed and the last aborts the transaction.

CNEW()
CNEW(a_conf,type_name,num)
Creates an array of "num" number of persistent instances of the type specified by the character string "type_name", in the configuration pointed to by "a_conf".

DELETE
Deletes transient or persistent instances.

DNEW()
DNEW(a_db,type_name,num)
Creates an array of "num" number of persistent instances of the type specified by the character string "type_name", in the database pointed to by "a_db".

SNEW()
SNEW(a_seg,type_name,num)
Creates an array of "num" number of persistent instances of the type specified by the character string "type_name", in the segment pointed to by "a_seg".

ID_OODB_TXN_BEGIN()
ID_OODB_TXN_BEGIN(read_only)
Begins a transaction with read-only access to objects if 1(true) is specified. If 0(false) is specified, it begins a transaction with write access allowed to objects. Nested transactions are not allowed in the generic interface.

ID_OODB_TXN_COMMIT
Commits the current transaction.

ID_OODB_TXN_ABORT
Aborts the current transaction.

DISTRIBUTION

Chief of Engineers
ATTN: CEHEC-IM-LH (2)
ATTN: CEHEC-IM-LP (2)
ATTN: CERD-L

Fort Belvoir, VA 22060-5516
ATTN: CECC-R

USA Engineer School 65473-5331
ATTN: USAES-DCD

Defense Technical Info. Center 22304
ATTN: DTIC-FAB (2)

9
+84
9/93

☆ U.S. GOVERNMENT PRINTING OFFICE: 1993—550-548/80057